Universidad Autónoma de Madrid

Escuela Politécnica Superior

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

PRUEBAS DE PRIMER ORDEN DE PROGRAMAS CONCURRENTES

Víctor de Juan Sanz
Tutor: César Sánchez Sánchez
Ponente: Juan de Lara

29 de junio de 2016

Universidad Autónoma de Madrid

Escuela Politécnica Superior

Double Degree in Computer Science and Mathematics

BACHELOR THESIS

FIRST ORDER PROOFS FOR CONCURRENT PROGRAMS

Víctor de Juan Sanz
Tutor: César Sánchez Sánchez
Ponente: Juan de Lara

June 29, 2016

PRUEBAS DE PRIMER ORDEN DE PROGRAMAS CONCURRENTES

Autor: Víctor de Juan Sanz
Tutor: César Sánchez Sánchez
Ponente: Juan de Lara

Computer Science
Escuela Politécnica Superior
Universidad Autónoma de Madrid

June 29, 2016

FIRST ORDER PROOFS FOR CONCURRENT PROGRAMS

Author: Víctor de Juan Sanz
Tutor: César Sánchez Sánchez
Ponente: Juan de Lara

Computer Science
Escuela Politécnica Superior
Universidad Autónoma de Madrid

June 29, 2016

Acknowledgements

Being Spanish my mother tongue language, I would like to take the freedom to express myself in such a great language.

Lo primero de todo quiero agradecer a César Sánchez, en representación de Imdea Software, la posibilidad de haber realizado con ellos el internship en verano y el Trabajo de Fin de Grado durante el primer cuatrimestre. El tiempo trabajado allí ha sido de gran ayuda, no sólo para la realización de este Trabajo de Fin de Grado, sino para mi elección de carrera profesional. Me ha permitido acercarme a la investigación académica, conocerla desde dentro y decidir en consecuencia. Me gustaría agradecer también a César Sánchez el tiempo dedicado y la paciencia mostrada conmigo. Ha sido un gran supervisor, animando mi creatividad para que este fuera mi trabajo, sin agobiar, simplemente dirigiendo, acompañando y aconsejando. Gracias también a Alejandro Sánchez por la ayuda, las explicaciones y las revisiones a lo largo de todo el internship.

Por otro lado, la realización de este trabajo me ha acompañado durante todo el año y durante el año muchas personas me han ayudado a llevarlo a buen puerto. Mi comunidad, mi equipo de voluntariado y mi equipo de catequistas, por liberarme de responsabilidades para que pudiera hacer un buen Trabajo de Fin de Grado. A mi familia y a mi comunidad por el apoyo mostrado, sobretodo durante el periodo de la solicitud de asignación fuera de plazo.

Por último, doy gracias a Dios por las oportunidades, las alegrías y los fracasos. Al fin y al cabo, *si el Señor no construye la casa, en vano se cansan los albañiles* (Salmo 126)

I beliebe in God, who reveals himself in the orderly harmony of what exists
Albert Einstein

Abstract

Abstract —

We study the uniform verification problem for infinite state processes. The problem consists on proving the parallel composition of an arbitrary number of processes running the same program satisfies a temporal property. As the general problem is too big for a bachelor thesis, we restrict our attention to concurrent implementations of sets using single linked list theory. We reduce the verification to the validity of formulas in this theory.

By validity we mean that certain property expressed as a formula of the theory holds. In our case, we prove that a list remains a list and that it is always ordered (both with independence of the number of processes executing on the same list). We could lock the whole list every time a process accesses it, but that is a very inefficient procedure. In this work we prove a grain-lock implementation.

Those proofs can be done using first order logic reasoning or model search. The approach chosen by Alejandro Sánchez and César Sánchez in his "Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures" (Sánchez, 2015) [1] was model searching. This work has been developed to complement "Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures" (Sánchez, 2015) [1] by providing a framework to prove the validity of formulas of the theories treated by them with first order logic.

As there are lots of proofs to be done, we decided to use an automatic theorem prover. *SPASS Version 3.5* [2] has been chosen because it has been used previously in the department.

We will also cover a brief discussion (using the results observed) about the benefits and the costs of this technique of verification. Is this way of verification always worth it?

Key words — Software verification, Parallel program verification, first order logic, verification conditions, safety, lifeness, temporal logic, automatic theorem prover (spass), fine grain-locking.

Resumen

Resumen — En este trabajo se estudia el problema de la verificación uniforme de problemas de estados infinitos. El problema consiste en probar que la composición paralela de un número arbitrario de procesos ejecutando el mismo programa satisface una propiedad temporal. Como es un problema demasiado ambicioso, nos hemos restringido en esta tesis al cliente más general que utilice una implementación de listas enlazadas con los procedimientos de insertar, eliinar y buscar. Es en este marco donde vamos a verificar el programa reduciéndolo a probar la validez de una formula en una teoría de listas.

Por validez nos referimos a que una cierta propiedad de la teoría se preserva. En este caso, vamos a provar que una lista siempre se mantiene como tal (es decir, no hay ciclos) y que se mantiene ordenada. Ambas propiedades se preservan con independencia del número de procesos ejecutando sobre la misma lista.

Para asegurar formalmente la satisfacción de estas condiciones, se puede utilizar búsqueda de modelos o razonamiento en lógica de primer orden. El método utilizado por Alejandro Sánchez and César Sánchez en su tesis "Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures" (Sánchez, 2015) [1] ha sido búsqueda de modelos. Este trabajo ha sido desarrollado para complementar "Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures" (Sánchez, 2015) [1] aportando un marco para probar algunas teorías utilizadas por ellos utilizando la lógica de primer orden.

Debido al gran número de demostraciones necesarias, hemos utilizado un demostrador de teoremas automático. *SPASS Version 3.5* [2] ha sido elegido debido a su previo uso en el departamento.

Además, incluimos una breve discusión a la luz de los resultados obtenidos sobre los beneficios y los costes de esta técnica de verificación. ¿Merece siempre la pena la verificación de software utilizando estos procedimientos?

Palabras clave — Verificación de Software, Verificación de programas paralelos, lógica de primer order, condiciones de verificación, safety, lifeness, lógica temporal, demostradores de teoremas automáticos (spass), fine grain-locking.

Acronym

ATP Automatic Theorem Prover. 6, 23

FOL First Order Logic. 2–4, 7–9, 11, 15, 31

LTL Linear Temporal Logic. 6

PROMELA PROcess MEta LAnguage. 6

SOL Second Order Logic. 6

SPL Simplified Programming Language. 7, 8, 14

VC Verification condition. 9, 16, 18, 31, 44

VCC Verifying Concurrent C. 6, 36

Contents

List of Tables

List of Figures

1 Introduction

Abstract In this chapter we will introduce this bachelor thesis. We will cover the motivation that led the development of this bachelor thesis and which objectives we pursue.

We will discuss its scope, the document structure and finally we will introduce the preliminaries, describing some concepts needed during the whole bachelor thesis.

1.1 Motivation

Last May 2nd a Japanese satellite was lost in space. This satellite cost $248 million. Research revealed the cause was a software error [3]. Last year, another software error was found in Boeing-787 aircrafts. Apparently, the plane's electrical generators fall into a failsafe mode if kept continuously powered for 248 days. [4] Fortunately this software error was discovered without relevant economic or human consequences. Between 1985 and 1987, six people died because of a software malfunction of an x-ray machine [5]. This are just a few example that justify software verification is a very important problem. One needs to be sure that the software being developed, in particular critical software, is correct. One would like it to work as expected with no bugs at all. There are some critical software as the ones developed for aeroplanes, spaceships, nuclear reactors which cannot have any errors while there are some other software which errors are more tolerable.

The usual approach to software reliability its testing. The normal way to verify and validate a software is running test to find, whenever is possible, all the errors. When the software is finished (or even while it is being developed) one can test it to check its correctness. How can one be sure that all functionalities have been proven? Maybe some cases were missed and some bugs have not been found so the software is not correct even though it passed all the test. As we said before, some systems can tolerate some level of incorrectness but there are some others which can not.

On the other hand, as a mathematician, I am very used to mathematical proofs of theorems

and I know logic is a very powerful tool. We wonder if it were possible to verify and validate software using those powerful tools. And the answer is affirmative. One can prove software correctness in a formal way. Software correctness can be proven in the same way as the Gauss theorem can be proven. One just need the appropriate framework, tools and of course, knowledge.

I have found this topic very useful and we wanted to explore a very formal verification of using some logical theories.

1.2 Objectives

The goal is to prove the correctness of an implementation of a concurrent linked list.[1]

We achieve to prove with mathematical certain the correctness of the program. We achieve to prove that a list is always preserved as a list and that it is always ordered, regardless the number of threads executing the program. Thus, there are 2 steps to prove. The first one is to prove that with just one process using the list, these properties are preserved always. The other one is with multiple processes using the same list. In particular, in a grain-lock implementation.

But to achieve any formal proof, we need some axioms as a basis. We can't define absolute truth, we can just proof that something is true, according to facts we already know. We can prove some theorem, but we must use the axioms as a starting point. So to achieve the verification, we need to define the axioms for the theory of linked list.

It is essential to build a framework in which this verification can be automated, so First Order Logic (FOL) will be used. Additionally, it is to be hoped that this FOL proofs can be generated, stored and checked by third-parties.

1.3 Scope

The scope of this thesis is to complement [1] with another approach formal verification. The authors has proven in [1] the correctness of an implementation of a concurrent linked list, but they used a different approach than the one chosen for this thesis.

1.4 Document Structure

Start by defining some necessary concepts to understand the rest of the thesis.

Once the reader is familiarized with some basic concepts, the state of the art is covered. We show some of the actual technologies even commercial products used nowadays.

Chapter 3 includes a preliminary section of what formal verification is and when and why a program can be formally verified and validated as correct. After that general preliminaries, the FOL theories used and the formalism necessary to do formal verification is explained. The implementation of the linked list is defined in this chapter.

[1] The specification of the implementation is on 3.3.1

Once the goal and the formalism is defined, the development can be fully understood. In Chapter 4, we describe the methodology used the tools developed or used.

Next, we show the results of the work. This chapter describes the list of axioms needed, an analysis of the FOL proofs generated and finally a time analysis.

Finally, the work is summarized in its conclusions. Does formal verification worth the try?

1.5 Preliminaries

Notation We assume the usual way of representing and working with FOL, this is

- **Symbols:** $\{), (, \implies, \iff, \vee, \wedge\}$
- **Quantifiers:** $\{\forall, \exists\}$
- **Constants:** $\{\top, \bot\}$ where we define $\top$ as *true* and $\bot$ as *false*.

One could consider $\exists x(P(x))$ as an abbreviation of $\neg(\forall x(\neg P(x)))$, but for better understanding we would use both quantifiers when needed. We could also use $(a \vee b)$ instead of $(\neg a \implies b)$ but, again, for the better understanding those abbreviations will be used. The same happens with $\top \equiv \neg \bot$, but it is clearer when we use both symbols and not just one of them.

Definitions We are going to define some very basic concepts, needed and used during the whole bachelor thesis.

Let X, Y be two sets of any dimension. A **function** denoted by $f : X \longmapsto Y$ is a map which takes elements from X and returns an element from Y. A **predicate** is a boolean-valued function, i.e., $P : X \longmapsto \{\top, \bot\}$ We call **arity** to the number of arguments a function or a predicate takes.

A **formula** is defined recursively as it follows: Constants, $\top, \bot$,predicates and functions are formulas. Let F_1, F_2 be two formulas. Then, $F_1 \implies F_2, F_1 \iff F_2, F_1 \vee F_2, F_1 \wedge F_2$ are formulas. We say a formula F is **satisfiable** iff there exists a model I that makes the formula true ($I \vDash F$). We say a formula F is **valid** iff for all interpretations I, $I \vDash F$. This 2 concepts are very important and they are very related. F is valid iff $\neg F$ is unsatisfiable.

A first-order **theory** is defined by the following components:

- Its **signature** Σ is a set of constants, functions and predicate symbols, where functions and predicates have a fixed arity.
- Its set of **axioms** $\mathcal{A}$ is a set of FOL closed formula in which only elements from Σ appear.

There are some important properties that a theory may have.

A theory Σ is **complete** iff for every closed $\Sigma-$formula σ we have $(\Sigma \vDash \sigma)$ or $(\Sigma\neg \vDash \sigma)$

A theory Σ is **consistent** if there is at least one $\Sigma-$interpretation. Equivalently, a theory Σ is consistent if $\Sigma \nvDash \bot$

If our theory is not consistent, we can have a formal proof of every formula, so we can prove any contradiction. We could prove that some program is both correct and incorrect at the same time, which gives no information. Thus **consistency is a fundamental property** of useful theories, like the ones used in this thesis.

In the other hand, completeness is very desirable, but may not be possible to achieve because of the incompleteness theorem by Gödel. It is not possible to have a complete and consistent theory that includes basic arithmetical truths. One could expect that the theory needed to prove programs correctness would not be complete, because of the inclusion of basic arithmetical truths. Normally, consistency is basic while completeness is only desirable.

Another property of theories is the **decidability**. We say a theory Σ is **decidable** if $\Sigma \vDash F$ is decidable, for every $\Sigma-$formula where a $\Sigma-$formula F is decidable if there is an **algorithm** that **always terminates** with "yes" if F is valid in Σ (Σ-valid) or "no" if F is not $\Sigma-$valid.

Decidability is a stronger property than completeness. As completeness, decidability is a very desirable property but because FOL (with no axioms) is undecidable in general, we may not have decidability in the theory we are working on.

Example: ***Theory of equality***

We are going to define the theory of equality, because it is the simplest first-order theory. The signature of the theory is:

$$\Sigma_e : \{=, a, b, c, ...\}$$

and it's axioms are:

Reflexivity: $\forall x.x = x$

Symmetry: $\forall x, y.x = y \implies y = x$

Transitivity: $\forall x, y, z.x = y \wedge y = z \implies x = z$

Function congruence: *For each function f of arity n*

$$\forall \overline{x}, \overline{y}. \left(\bigwedge_{i=1}^{n} x_i = y_i \right) \implies f(\overline{x}) = f(\overline{y})$$

Predicate congruence: *For each predicate P of arity n*

$$\forall \overline{x}, \overline{y}. \left(\bigwedge_{i=1}^{n} x_i = y_i \right) \implies P(\overline{x}) \iff P(\overline{y})$$

> *This 2 "axioms" are not axioms but* ***axiom schemas****, because there is one axiom for each function f or predicate P.*

FOL with equality is a decidable theory as Leopold Löwenheim proved in 1915 [6]. It is also consistent and complete.

2

State of the art

Abstract Formal verification is needed in some parts of software industry. We present some examples.

There are some tools that aims to offer solutions to the need of formal verification. One of the tools covered is developed by Microsoft, which illustrates that this topic is not irrelevant. Additionally, we cover different ways this *formal verification* can be attempted.

2.1 The need of formal verification

As it was shown in the introduction in Section 1.1 there are some systems in which software errors are totally inadmissible. People developing software for critical systems need an effective way to check the correctness of this software. In addition, they need some guarantee that the compiler generates an executable which exhibits **exactly** the same behaviour as the source program. In order to solve this issues there are some tools which have been developed.

2.1.1 Solutions

Compcert: School of Computing in the University of Utah claims [7]

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than 325 previously unknown bugs to compiler developers. Moreover, every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs.

So there is a real need of a C verified compiler and that is what **Compcert** intends to be. Comcpert can compile programs using almost all of the ISO C90 / ANSI C. The project began in 03/2008 and is still under development. Last version was released in 12/2015. This research group has formally verified compilers for functional languages [8].

VCC: Verifying Concurrent C (VCC) is a tool developed by Microsoft Research which allows to develop Verified C Code [9]. The aim is to offer a tool to critical developers in which the formal verification needed in critical software is integrated and is easy to use. This project encourages the use of formal verification among developers.

VCC is sound, which means that if VCC verifies a program, it really is correct, with 2 possible problems. VCC is not verified itself, which means it can have bugs. Additionally, the compiler used bt VCC is verified This tool can be downloaded at reseach.microsoft.com

2.2 Types of formal verification

There are basically two different ways of formally verify a program. Both of the consist on translating the program into a logical formula and try to prove it valid. The first way is to search models and the other one is try to obtain a formal proof. Model searching is easily automatizable but formal proofs can not be automatized for some logic, as Second Order Logic (SOL).

2.2.1 Model checking

The approach of model checking is the one used in "Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures" (Sánchez, 2015) [1]. Another tool that is based on model checking is Spin.

Spin: Spin targets the efficient verification of multi-threaded software. The tool checks the logical consistency of a specification and reports on deadlocks, race conditions, different types of incompleteness, and unwarranted assumptions about the relative speeds of processes. Spin provides direct support to C code by including a tool which translates C code to PROcess MEta LAnguage (PROMELA). Spin also provides a Linear Temporal Logic (LTL) model checking system.

Two examples of inspiring applications of Spin in the last few years are the verification of the control algorithms for the new flood control barrier built near Rotterdam. The verification work was carried out by the Dutch firm CMG (Computer Management Group) in collaboration with the Formal Methods group at the University of Twente.

2.2.2 Formal proof

The other approach is to use an Automatic Theorem Prover (ATP) which allows automatic proving. With the theory and the formula, the ATP tries to find a proof. As it was explained in Section 1.5, we may need to use a theory which is not decidable, which means that the ATP may be searching for the proof for ever.

This is the approach we have taken in this thesis. There are different ATP, but in all of them, a framework must be build in order to use them. Essentially, they are theorem prover with no interface difference apart from the syntax. Internally, they search the proofs in different ways. Some examples of ATP are Vampire[10], Issabelle [11] and Spass [2]

3 Design

Abstract In this chapter we define the way that formal verification can be achieved, for which first we define some notation and definitions. After some generalities about formal verification, we define more concrete aspects of the formal verification we aim to achieve, such us the linked-list theory used.

3.1 Program correctness

We are finally ready to apply this concepts to a real word problem. In this bachelor thesis we apply those concepts to prove some properties of programs. The remaining task is to define the framework and the conventions we use to formally prove properties of programs.

The way we approach to assess correctness is by proving properties. There are **liveness**, **safety** and **functional** properties. Safety properties refer, informally, to "bad things never happens". Proving *variable x is never 0* is a safety property. Proving valid this property can assure that a division by zero error will never occur. Whether a program finishes or not is a liveness property, and producing an output for a concrete input is a functional property.

These properties are written in some logic. Liveness properties require the use of temporal logic but we restrict ourselves to use safety properties so no machinery for temporal properties is needed. As the properties are expressed formally in FOL, it is necessary to define a formal representation of a program.

3.1.1 Formal Representation of a Program

This Simplified Programming Language (SPL) and its formal representation is the language chosen to write the programs to be formally verified. It has been chosen by [1] because its simplicity and expressiveness in order to write concurrent programs. Because its simplicity it is a great option to do formal verification with it.

Preliminaries (Notation, Definition)

The semantics of programs are given as a sequence of states. The pc variable (**Program counter**) has the information of the line to execute next. Additionally, there are the steps of the program. Each step modifies the state of the program by modifying the values of the variables including the pc. A step can be easily expressed in FOL using **post-state** variables, which are the new values the variables will have after the execution of the line. The formula gathering the information of the execution of a line, using the pcand pc' (the post-state pc) and all the other variables is called **transition relation**.

Possible instructions

In order to express correctly the transition relation corresponding to certain line, we need to know how to translate program statements into FOL. As we are going to work with programs used by more than one thread, we need one program counter for each thread executing. We parametrize the program counter by thread identifiers. That is, we define the **program counter** as a **function** that given a thread, returns its program counter. We could have introduced one variable per thread. It would be an equivalent formulation.

We proceed to define in general terms how to build transition relation for a SPL statement. We only show two types of statements. All the others statements are defined in a similar way. A complete definition can be found in [12]. For these definitions we use the letter T to refer a thread.

Assignments: The transition relation for a variable assignment consists of the update of the program counter for the running thread and the corresponding modification to the variable being assigned.

Statement	Transition relation
l_1 : $\mathtt{v} := 2$	$pc(T) = l_1 \wedge pc'(T) = l_2 \wedge \mathtt{v}' = 2$
l_2 : $\cdots$	

Loops: We consider the only loop statement available in SPL which executes the statements in the body as long as the loop condition holds.

Statement			Transition relation	
ℓ_1	:	**while** c **do**	$(pc(T) = \ell_1 \wedge \ \mathtt{c} \wedge pc'(T) = \ell_2) \vee$	
ℓ_2	:	$\cdots$	$(pc(T) = \ell_1 \wedge \neg\mathtt{c} \wedge pc'(T) = \ell_{n+1})$	for line ℓ_1
$\vdots$				
ℓ_n	:	**end while**	$pc(T) = \ell_n \wedge pc'(T) = \ell_1$	for line ℓ_n
ℓ_{n+1}	:	$\cdots$		

3.1.2 Partial correctness (Safety)

A function (or the whole program) is **partially correct** if whenever the function's precondition is satisfied on entry, its postcondition is satisfied when the function returns (if it ever does). We

present the **inductive assertion method** for proving partial correctness.

Let φ be the FOL property to study. The procedure is the following: First each function is reduced to a finite set of FOL formulae called **Verification condition (VC)**. This reduction is done with the basic reducing cases we studied in 3.1. The goal is to prove that φ is valid in every state of the execution. **Induction** is the methodology used. First, we assert φ is valid before the program starts (induction base). Then, we assume φ in the precondition and prove φ' valid (induction step) for every possible transition.

This method is not complex to understand but it requires a lot of work even for simple programs. We illustrate this method with an example in A.

3.2 Parametrized systems

The correctness of a program executed by just one thread it is an easier problem because the program runs sequentiality. Multiple threads executing the same program is a different and more difficult problem to solve. An unbounded number of threads executing is another important and difficult extension. If the number of threads is bounded, one could unroll the formula for all the threads in the problem. As this is the usual scenario, we focus the unbounded case.

We are going to study those cases. To do so, we need to parametrize the program executed by multiple threads. Typically we use i,j,k_0,k_i for threads identifiers.

Arbitrary number of threads

For example, the web servers may not have a bound of the number of clients they can accept. Can we prove correctness when an unbounded number of processes are using the same global variables?

A recent research "Parametrized Invariance for Infinite State Processes" (Sánchez and Sánchez, 2013) [13] has proven a very important result. We will present this result new as it is fundamental for this work. We will not formally prove any of the results proven in [13].

Before we enunciate the theorem, we need some previous concepts. We need to extend the concept of support to parametrized formulas.

Definition 3.2.1 **Support** Let ψ, A and B be parametrized formulas, and let S be the set of possible substitutions from the set of parametrized variables in ψ ($Var(\psi)$) into the set of parametrized variables of $(A \to B)$ ($Var(A \to B)$). We say that ψ supports $(A \to B)$, whenever

$$\Big(\big(\bigwedge_{\sigma \in S} \sigma(\psi)\big) \wedge A\Big) \to B \qquad \text{is valid}$$

We use $\psi \rhd (A \to B)$ as a short notation for $\Big(\big(\bigwedge_{\sigma \in S} \sigma(\psi)\big) \wedge A\Big) \to B$. Note that if $S' \subseteq S$ is a subset of the substitutions, and

$$\Big(\big(\bigwedge_{\sigma \in S'} \sigma(\psi)\big) \wedge A\Big) \to B \qquad \text{is valid}$$

then

$$\Big(\big(\bigwedge_{\sigma\in S}\sigma(\psi)\big)\wedge A\Big)\to B \qquad \text{is also valid}$$

Essentially, if one succeeds in proving the validity of a formula obtained by removing some of the conjuncts from the antecedent, the validity of the full formula is preserved. Hence, in practice, it is enough to consider only some of the partial substitutions to show that a support formula is valid.

Theorem 3.2.2 (Bound an arbitrary number of threads)**.** *Let φ be a thread-parametrized formula, where $\overline{k} = Var(\varphi)$. Let τ be a transition of P and Θ_{param} the initial condition of P.*

To show that P satisfies $\Box\varphi$ (that is, φ is an invariant of P):

$$\frac{\begin{array}{lrcll}
\mathsf{S1}. & \Theta_{\mathsf{param}}(\overline{k}) & \rhd & \varphi & \\
\mathsf{S2}.\ \varphi\rhd & \tau^{(i)} & \to & \varphi' & \textit{forall } \tau \textit{ and all } i\in\overline{k} \\
\mathsf{S3}.\ \varphi\rhd & \Big(\bigwedge_{x\in Var(\varphi)} j\neq x\wedge\tau^{(j)} & \to & \varphi'\Big) & \textit{forall } \tau \textit{ and one fresh } j\notin\overline{k}
\end{array}}{\Box\varphi}$$

Using this powerful result, we have reduced an arbitrary number of processes sharing the same variables to a finite number of threads sharing the variable. The proof of this result can be found in "Parametrized Invariance for Infinite State Processes" (Sánchez and Sánchez, 2013) [13]. We will refer to S1 as **initiation** because it depends on the initial condition. S2 will be referred as **self-consecution** because it captures the execution of one of the threads mentioned in the formula. Finally, S3 will be referred as **others-consecution** because it captures the execution of threads which do not appear in the formula. The example included in appendix A illustrate the concept of support.

3.3 A Theory of Linked Lists

3.3.1 Description

In order to work with linked lists in a context with multiple thread using the same list there are two possible approaches. A thread could lock the entire list, work with the list and then release the lock. There could be some optimizations in this approach, such as a writer-reader system. However, this is extremely unefficient although it could more secure in terms of preventing deadlocks. The other approach is locking and unlocking each node of the list, so multiple threads can work simultaneously using the same list as long as they do not need to use the same node. This approach is called lock-coupling lists.

Definition 3.3.1 **Lock-coupling linked list** A lock-coupling concurrent list is a concurrent data type that implements a set by maintaining in the heap an ordered single-linked list with non-repeating elements. Each node in the list is protected by a lock which guarantees that a single thread can access the node at the same time.

The way a thread iterates over the list is the following. The thread acquires the lock of the

node that it visits and after that tries to acquire the next node. The first lock is only released after the lock of the second node has been successfully acquired. This technique of protecting cells with locks (instead of protecting the whole data-structure with a single coarse-grain lock) is known as **fine-grained locking**.

The nodes of a concurrent lock-coupling list are instances of the following *ListNode* class:

$$\textbf{class}\ \mathit{Node}\ \{\mathit{Elem}\ \ \mathit{data};\ \mathit{Addr}\ \ \mathit{next};\ \mathit{Lock}\ \ \mathit{lock};\ \}$$

Where the fields are:

- *data*: the value stored in the node. This field is also used to keep the list ordered.
- *next*: a pointer that stores the address of the next node in the list.
- *lock*: the lock protecting the node.

We assume that the operating system provides the atomic operations *lock* and *unlock*.

We will use **ghost variables** which are variables that are not present in the program but are added to aid in the verification process. The implementation of concurrent lock-coupling lists has 3 global variables. Two of them are global addresses *head* and *tail*, and one ghost global variable *reg*. The variable *head*, an address points to the first node of the list which has the lowest possible value ($-\infty$). The variable *tail*, an address points to the last node of the list which has the lowest possible value ($+\infty$). Finally, the variable *reg*, a set of addresses, is used to keep track of the portion of the heap whose cells form the list. In Figure 3.3.1 we present the code of the implementation chosen.

There are three procedures, Search, Insert and Remove which traverses through the list the way it was explained.

3.3.2 TL3

To prove verification conditions generated in the proof of invariants of lock-coupling lists we need a theory of lists to work with, and axioms in order to prove FOL formulas.

Theory of Linked Lists with Locks: TL3, is the theory we use for describing linked-list heap memory layouts. TL3 is a multi-sorted first-order theory. It is multi-sorted because it has multiple types for its variables (address, element,...). It is a first-order theory because only variables are quantifiable, as in unsorted FOL.

In this section we briefly present TL3. A more complete and formal definition of TL3 can be found in "A Theory of Skiplists with Applications to the Verification of Concurrent Datatypes" (Sánchez and Sánchez, 2011) [14] and [1, p. 6.2].

Although some functions are originally defined [1] in suffix notation (*next*,*data* and *lock* fields), preffix-notation has been used to describe the theory. The reason for this modification is to be consistent with the syntax of Spass. Furthermore, we use subset of TL3. In the same way FOL can be expressed with $\neg, \vee$ but sometimes $\Longrightarrow$ is included but $\Longleftrightarrow$ is not, a few functions of TL3 have not been used because they can be expressed using others functions in the theory. We proceed to describe the subset of TL3 used.

```
global
  addr  head;  addr  tail;
  setaddr  region;
assume
    region = {head, tail, null}
 & head ≠ tail  &  head ≠ null  &  tail ≠ null
 & head→data = −∞  &  tail→data = +∞
 & head→next = tail  &  tail→next = null

Procedure MGC()
  elem  e
begin
1: while true do
2:     e := HavocListElem()
3:     non deterministic choice:
4:       call Search(e)
       or
5:       call Insert(e)
       or
6:       call Remove(e)
7: end while
   end procedure
```

```
Procedure Insert(e)
  addr  prev
  addr  curr
  addr  aux
begin
23: prev := head
24: prev→lock()
25: curr := prev→next
26: curr→lock()
27: while curr→data < e do
28:     aux := prev
29:     prev := curr
30:     aux→unlock()
31:     curr := curr→next
32:     curr→lock()
33: end while
34: if curr ≠ null  &  curr→data > e then
35:     aux := malloc(e, null, #)
36:     aux→next := curr
37:     prev→next := aux
        region := region ∪ {aux}
38: end if
39: prev→unlock()
40: curr→unlock()
41: return
    end procedure
```

```
Procedure Search(e)
  addr  prev
  addr  curr
  addr  aux
  bool  found
begin
 8: prev := head
 9: prev→lock()
10: curr := prev→next
11: curr→lock
12: while curr→data < e do
13:     aux := prev
14:     prev := curr
15:     aux→unlock()
16:     curr := curr→next
17:     curr→lock()
18: end while
19: found := (curr→data = e)
20: prev→unlock()
21: curr→unlock()
22: return found
    end procedure
```

```
Procedure Remove(e)
  addr  prev
  addr  curr
  addr  aux
begin
42: prev := head
43: prev→lock()
44: curr := prev→next
45: curr→lock()
46: while curr→data < e do
47:     aux := prev
48:     prev := curr
49:     aux→unlock()
50:     curr := curr→next
51:     curr→lock()
52: end while
53: if curr ≠ tail  &  curr→data = e then
54:     aux := curr→next
55:     prev→next := aux
        region := region \ {curr}
56: end if
57: prev→unlock()
58: curr→unlock()
59: return
    end procedure
```

Figure 3.1: Code of the implementation chosen.

TL3 is a composition of theories. The **sorts** used among this theories are: cell (representing the nodes of the list), elem (representing elements), addr (representing address), tid (representing thread id), mem (representing the memory also called heap, represented as maps of addr to cell), path (representing a finite sequence of address), settid, setaddr, setelem to represent sets of tid,addr or elem respectively.

For each sort, there is a theory containing its constants, functions and predicates. There is one more theory, Σ_{Bridge} is a *bridge theory* containing auxiliary functions, for example, that allow to map paths of addresses to set of addresses, or to obtain the set of addresses reachable from a given address following a chain of *next* fields.

3.3.3 Signature

We proceed to describe the signature of each theory, listing the sorts used and explaining its functions, predicates and constants. Every theory includes the equality theory 1.5

Σ_{tid} **:** The sort used is tid. The "no-thread" value is represented with $\oslash$. Apart from the equality theory, this theory does not have any other predicates or functions.

Σ_{elem} **:** The sort used is elem. There is a total order which allows to order every set of elem. In addition, this sort is upper and lower bounded. The top block contains the functions and the lower block lists the predicates.

highestElem	elem	Maximum value an elem can take.
lowestElem	elem	Minimum value an elem can take.
ls_elem	elem ×elem	Total order relation between elem.

Σ_{cell} **:** The sorts used are cell, elem, addr, tid.

mkcell	elem × addr × tid → cell	Constructor
next	cell → addr	Getter of *next* field
data	cell → elem	Getter of *data* field
lockid	cell → tid	Getter of *lockid* field
lock	cell × tid → cell	Construct a new cell with *data* and *next* values of the given cell, using the tid for the *lockid* field.
error	cell	Constant value used to model incorrect memory deference.

The function *unlock* could be considered. Actually, [1] includes it in the theory but it has not been included in this work. The reason is justified because to *unlock* a cell is equivalent to *lock* a cell with $\oslash$ value.

Σ_{mem} **:** The sorts used are mem,cell and addr.

null	addr	Null address
rd	mem × addr → cell	Models memory deference. Returns the value from the mem the cell stored in the addr.
upd	mem × addr × cell → mem	Creates a new mem from the given one

A function related with mem theory is *malloc*, used in the `insert` procedure. The function *malloc* does not belongs to SPL or TL3 but it can be translated as a conjunction of assignments and assignments are allowed in both theories. The function *malloc* returns a new fresh address different to every other address in use, so the *freshaddr* returned by *malloc* is not equal to *head*, nor *tail*, etc. *malloc* formal representation correspond to a big conjunction of all the formulas stating *freshaddr* is not equal to addr, for all addr appearing in the formula (except itself).

Σ_{setaddr} **:** It models the usual set theory. We preferred a prefix version of each function and predicate to be consistent with Section 5.1.

The intersection function and the subset predicate have not been included, even tough [1] uses them. They were not used because they were redundant.

emptyset	setaddr	Empty set
singl	addr → setaddr	Constructor of a single-element set.
Union	setaddr × setaddr → setaddr	
setDiff	setaddr × setaddr → setaddr	
in	addr × setaddr	

Σ_{setelem} **:** Again, it models the usual set theory. The signature is described in Table 3.3.3.

emptysetElem	setaddr	Empty set
singlElem	elem → setaddr	Constructor of a single-element set.
UnionElem	setaddr × setaddr → setaddr	
setDiffElem	setaddr × setaddr → setaddr	
inElem	addr × setaddr	

Σ_{settid} **:** The signature is described in Table 3.3.3.

emptysetTh	setaddr	Empty set
singlTh	addr → setaddr	Constructor of a single-element set.
UnionTh	setaddr × setaddr → setaddr	
setDiffTh	setaddr × setaddr → setaddr	
inTh	addr × setaddr	

Σ_{Bridge} **:** This theory is much more extensive in [1]. However, aiming for simplicity, we do not include every function and predicate because we do not use them in our proofs. The only function used in the proofs is:

addr2set	mem × addr → setaddr	Returns the set of addr reachable from the addr given.

4

Development

Abstract In this chapter, we describe the practical work. Until now, we just have defined the theoretical foundations.

Here we will expose the rigorous methodology used to mathematically perform our verification. To achieve our goals, we have developed some tools which are described here too.

We will end the chapter defining precisely the goal and the logic formulas needed to prove valid in order to claim that the program is verified.

4.1 Methodology Used

4.1.1 Process, Tools Used and Tools Developed

Spass

Spass [2] is an automated theorem prover for first-order logic with equality. Spass receives a FOL formula and tries to prove it valid. Running SPASS on such a formula results in the final output "*SPASS beiseite: Proof found.*" if the formula is valid or "*SPASS beiseite: Completion found.*" if the formula is not valid. Because validity in first-order logic is undecidable, SPASS may run forever without producing any final result. This last comment is a very important issue because some proofs have taken hours and one could not know for sure if Spass would eventually stop or run forever. As a curiosity, the longest time Spass was left running was 82 hours and it stopped because a proof was found.

Leap

Leap is a tool for the verification of concurrent data types and parametrized systems executed by an unbounded number of threads which manipulate mutable shared data in the heap.

Leap receives as input a concurrent program description and a specification and automatically generates a finite set of verification conditions which are then discharged to specialized decision procedures. The validity of all discharged verification conditions implies that the program executed by any number of threads satisfies the specification. Currently, Leap includes not only decision procedures for integers and Booleans, but it also implements specific theories for heap memory layouts such as linked-lists and skiplists.

Spolv

Spolv is a tool implemented as part of this project to aid Spass proving VC. Spolv implements the process of converting the VCs generated by Leap to Spass syntax, splitting different conjunctions in different files, calling Spass to try to prove and process and storing the results has been automatized basically in python combined with bash scripts. Additionally, Spass reads prefix syntax while Leap uses prefix and infix syntax.

In addition, Spolv does some reduction of the formulas so Spass can finish in a reasonable time. This is needed because Spass does not use any information or tactics to decide which axiom should be used first.

Process of Tool Cooperation

We describe now the way in which the tools cooperate. The process followed is shown in Fig. 4.1.

The first step consists on translating the program into VC. This is done by Leap. In the case of verifying single linked list, Leap generates at least 2 VC for each transition of the program. The first VC corresponds to self-consecution, and the second corresponds to others-consecution. The initial transition is also generated. The process is repeated 6 times, one for each candidate invariant of the problem (4.2).

Once the VCs have been generated, the goal is to prove all of them using Spass. As the syntax of Spass ([15]) is not Leap syntax some parsing is needed. For example, Spass uses prefix notation Leap uses infix notation for binary functions. In order to solve this, it was necessary to learn Ocaml using [16]. Leap was forked so it could write the VCs in prefix notation. Some other Leap functionalities had to be changed to make Leap output compatible with Spass input.

In addition, one Spass problem has to be created for each VC. The axiom list for each problem is determined as an argument because not every Spass problem needs the same axioms. This is explained further in section 5.2.

Because of Spass lacks of tactics and the relatively large size of the axiom list, some very easy proofs could take a long time. For transitions which can be proven using very simple reasoning using pcs,Spass could take minutes. In order to improve Spass performance, we decided to divide the problem upfront.

Let φ be the VC to prove. Consider for example the following VC

$$\varphi : \quad pc(i) = l_j \rightarrow head \neq tail$$

While proving the VC for the thread i, $pc(i)$ has a value which may be l_j or not. In both cases,

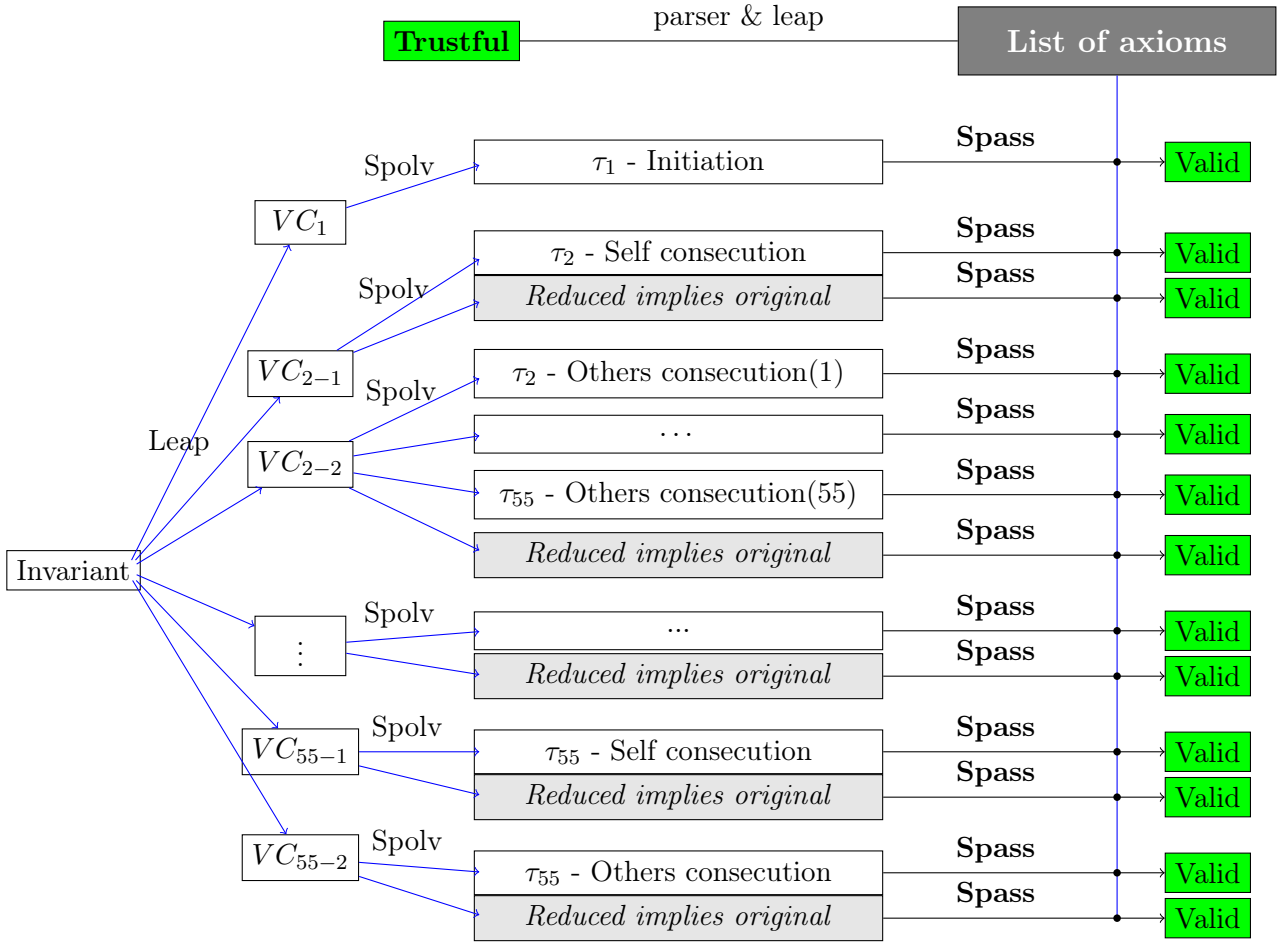


Figure 4.1: Process graph.

resolution can be applied to prove equivalently

$$\psi : \quad head \neq tail$$

This resolution can be applied in self-consecution and in others-consecution. This resolution step is done by Spolv. Some other obvious resolution is also performance at this stage, such as $\top \wedge \top \equiv \top$ and other tautologies.

Now Spass can prove a simpler problem requiring much less time. To make sure that both problem are equivalent and that the conversion was sound, we generate another Spass problem. This Spass problem aims to prove $\psi \rightarrow \varphi$. In order to prove φ, modus ponens is used:

$$\frac{\begin{array}{l}\psi \\ \psi \rightarrow \varphi\end{array}}{\varphi}$$

Even though two Spass problems have to be solved instead of one, in 5.3 the reader can see the benefits of applying this method. The reason is that φ is a complex problem for Spass. However, ψ is a much simpler problem because it does not include any reasoning about program counters. In addition, the problem $\psi \rightarrow \varphi$ is tautologic in most of the transitions because it adds conjectures in the antecedent of an implication, which cannot make invalid a valid formula. Plus, the list of axioms needed to prove ψ is smaller where compared to the axiom list needed to prove the original problem.

The generated problem ψ will be called the ***Reduced Spass problem*** and $(\psi \rightarrow \varphi)$ will be called the ***Reduced implies original***. φ problem will be called the ***Original Spass problem***

The *Reduced Spass problem* should not have any *pc* involved. How can the *pc* be removed while proving others-consecution? Again, more Spass problems are generated. In this case, 55 new problems are generated and using modus ponens:

$$\frac{\begin{array}{l}\psi_1 \\ \psi_2 \\ \vdots \\ \psi_{55} \\ (\bigwedge_i \psi_i) \rightarrow \varphi\end{array}}{\varphi}$$

In this case again, the Spass problem $(\bigwedge_i \psi_i) \rightarrow \varphi$ is tautologic most of the time, but Spass takes much more time than before because of the problem size.

4.1.2 Auxiliary Tools

Ocaml- parser

All the process described above require an axiom list. Getting the axioms needed to prove all the transitions was the most difficult work. Typically, we provided Spass with an axiom list with which all VCs were proven. In the process of checking the generated proofs and verify they were correct, we found some inconsistencies. This inconsistencies were always consequence of incorrect axioms. Because of human ingenuity, it was very difficult to ensure absolute validity

of the axioms.[1] In order to solve this, we decided to validate every axiom using Leap. This procedure does not assure absolute validity to the axioms but helped a lot to find incomplete axioms.

As there are several axioms, a parser was implemented to automatize the parsing files from Spass syntax to Leap. The parser was implemented in Ocaml so Ocaml skills could be improved. Additionally, *Ocamllex* [17] and *Ocamlyacc* [18], Ocaml variants of the C tools studied during the bachelor degree (*Proyecto de Autómatas y Lenguajes*) have been used to implement the parser. The functionality of translating from Spass syntax to LaTeX syntax has also been included to generate the axiom list 5.1

4.2 Linked list

We manage to prove that the implementation proposed at 3.3.1 of a concurrent lock coupling linked-list always preserves the list structure. There are some conditions to assure that a set of nodes is a list. An order has to be preserved, *head* and *tail* must keep the properties of its definition. We gather all the necessary conditions in formula (4.1):

$$\mathsf{list} \stackrel{\text{def}}{=} \begin{cases} null \in reg \wedge reg = addr2set(heap, head) \wedge head \neq tail & \wedge & \text{(L1)} \\ heap[tail].next = null \wedge tail \neq null \wedge head \neq null & \wedge & \text{(L2)} \\ heap[head].data = -\infty \wedge heap[tail].data = +\infty & \wedge & \text{(L3)} \\ Ordered(heap, head, tail) & & \text{(L4)} \end{cases} \tag{4.1}$$

(L1) establishes that *null* is in *reg* and that *reg* is exactly the set of addresses reachable in the *heap* starting from *head*, which ensures that the list is not circular. (L2) and (L3) express some sanity properties of the sentinel nodes *head* and *tail*. Finally, (L4) express the fact that the list is ordered.

We claim that the formula list is an invariant of this implementation. The full proof can be found later in 5.2. Although this may seem easy, human ingenuity makes it hard. In "Decision Procedures for the Temporal Verification of Concurrent Lists" (Sánchez and Sánchez, 2010) [19] the authors whoen that list is not an inductive invariant. We need others invariants as support.

Invariants of the implementation

As these are not principal but auxiliary invariants, we just provided here a brief description of them. These auxiliary invariants are needed to prove list and a full description of each formula appear in D.

disj: This invariants provides the information needed when inserting a new element. Two different *malloc* invocations return two different addresses, so there is not possible to insert the same address twice in the list.

[1] Axioms should be universally valid (not just valid in TL3 theory) so they could be used for other theories that Leap can work with.

region: These invariants refers to the information about to which addresses are in the ghost variable *reg* var, that is, which addresses are reachable from *head*.

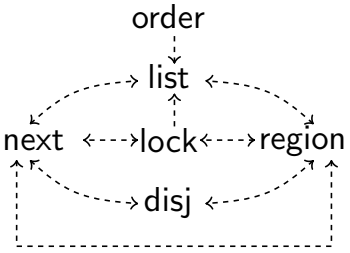
lock: These invariants refer to the information provided by the locks. The *lock* acquired at line l_i is not unlocked until line l_{i+j}, so at lines l_k $k \in (i, i+j)$ the *lock* should be locked by the running thread. This is a very important invariant because it assures that concurrent access are safe and therefore it is well implemented.

order: These invariants refer to the information related to the order of the list.

next: These invariants refer to the information provided by *next* statements. If at l_i, *curr* := *prev* $\rightarrow$ *next*(), is executed *curr* still has the value *prev* until a new assignment to *curr* is done. This invariant encodes the information about the relation between addresses.

Dependencies between invariants

The only inductive invariant is order. The rest of the auxiliary invariants need some other invariants to prove them. In figure 4.2 the graph of dependencies is shown. As it is shown, the invariant disj is not directly needed to proof list, but it is needed as support to proof both next and region.



One could think that it is not sound to prove lock using region as support and prove region using lock as support. This apparent circularity is an important point for the proofs. We claim this apparent circularity is not a circularity and it is sound.

Let φ,ψ be two invariant candidates of a program such that the initiation step is valid for both of them. It is sound using φ as support for proving ψ and using ψ as support for proving φ. More formally, we show it as a lemma with its proof.

Lemma 4.2.1. *Let φ and ψ be two candidate to invariants of a program.*

Let θ be the initial condition and let initiation hold for φ and ψ.

$$\theta \rightarrow (\varphi \wedge \psi)$$

Let ψ be a support of φ and φ be a support of ψ. Then, for every transition τ_j in the program we have:

$$\forall j, \tau_j \wedge \varphi \wedge \psi \rightarrow \psi'$$

$$\forall j, \tau_j \wedge \varphi \wedge \psi \rightarrow \varphi'$$

Then, *ψ and φ are invariants.*

Proof: Proof by contradiction: Let ψ not be an invariant. Thus, there is at least one state j in which ψ_j does not hold. Lets take j as the first counterexample. By hypothesis, $\psi(0)$ (initiation) is valid, so $j > 0$.

Lets take the transition τ_{j-1}. As j is the first counterexample, in state $j-1$ both ψ, φ hold. Thus, by hypothesis, $\psi_{j-1} \wedge \tau_{j-1} \wedge \varphi_{j-1} \rightarrow \psi_j$ is valid. This is a contradiction because ψ_j was not valid. □

5 Results

Abstract This chapter describes the set of axioms needed to prove the target verification problem and the justifies the relevance of the axioms. We will cover briefly the generated proofs and some complexity results. Finally, we report a time analysis, comparing this ATP to model checkers and the different inner steps of the methodology.

5.1 Axioms

We introduce now the set of relevant axioms needed to prove all the invariants. There are some secondary axioms needed by Spass that have been omitted. The omitted axioms refer:

- to the sorting (Spass is not multi-sorted as the theory we work with. It is necessary to define that an addr is not an elem, an elem is not an addr, etc)
- to constants (Spass does not include arithmetic, so 0,1,... must be defined as unique 0-ary functions specifying that $0 \neq 1, 0 \neq 2$,...)
- to the local and global variables of the program (Leap is quantifier free but Spass needs to quantify every variable)

The set of axioms has been divided in groups. The division is as follows

Common

This are some common axioms, included in every Spass problem generated.

Table 5.1 reports the uses of each axiom when proving each invariant.

Axiom next-error–is–null: *The field next of the error* cell *is the null address.*
Here we present the formula:

$$\forall * \quad next(error) = null \tag{5.1}$$

axiom	disjoint	locks	nexts	order	preserve	region
next-error–is–null	0	1	194	142	0	8

Table 5.1: Uses of the common subset of axioms.

Arithmetic

This subset of the axioms refers to the arithmetical axioms needed at the proofs.

Table 5.2 reports the uses of each axiom when Spass proves each invariant.

Axiom nums-are-different: *All numbers are different one another.* [1]
Here we present the formula:

$$\forall * \quad (0 \neq 1) \wedge (0 \neq 2) \wedge ... \wedge (1 \neq 2) \wedge ...(54 \neq 55) \tag{5.2}$$

axiom	disjoint	locks	nexts	order	preserve	region
nums-are-different	118	1	109	1	0	44

Table 5.2: Uses of the arithmetic subset of axioms.

Set theory

This set of axioms models some rudiments of set theory. As there are 3 types of sets in TL3, this subset of axioms are replicated for $\Sigma_{\mathsf{setaddr}}, \Sigma_{\mathsf{setelem}}$ and Σ_{settid}.

Table 5.1 reports the uses of each group of 3 axioms in each invariant.

Axiom union-def: *Let x be a generic, and $se1, se2$ set of generics. Then x belongs to the union of $se1, se2$ iff x is in $se1$ or $se2$.*
Here we present the formula:

$$\forall * \quad ((in(x, se) \vee in(x, se2)) \iff in(x, union(se, se2))) \tag{5.3}$$

Axiom set-equal: *Two set are equal iff there not exists a generic that is in one set but not in the other. This axiom is called in literature [20]* ***set extensionability****.*
Here we present the formula:

$$\forall * \quad [\nexists\, a\, (in(a, se2) \iff in(a, se1))] \iff se1 = se2 \tag{5.4}$$

[1] This axiom is necessary because Spass only includes the equality theory. Thus, 0,1,...,55 are 0-ary functions that could potentially be the same number.

Axiom union-commutative: *The union of set is a commutative operation.*
Here we present the formula:

$$\forall * \;\; union(se, se2) = union(se2, se) \tag{5.5}$$

Axiom in-set–def: *Let a, b be generics and se a set of generics. If a is not in the set but b is in the set, then a and b can not be equal.*
Here we present the formula:

$$\forall * \;\; ((\neg\, in(a, se)) \implies in(b, se) \implies (\neg\, b = a)) \tag{5.6}$$

Axiom a–in–singl-a: *Let se be the singleton set built from the generic element a. Then, there are no other address different from a in se. We call $se = singl(a)$.*
Here we present the formula:

$$\forall * \;\; (((\neg\, a = b) \implies (\neg\, in(b, singl(a)))) \wedge in(a, singl(a))) \tag{5.7}$$

Axiom emptySet-is-empty: *The generic empty set does not contain any generic.*
Here we present the formula:

$$\forall * \;\; (\neg\, in(a, empty)) \tag{5.8}$$

Axiom set-exten-inv: *The inverse of set extensionability, described above (5.1).*
Here we present the formula:

$$\forall * \;\; \big((\forall a \;\; in(a, se1) \iff in(a, se2)) \implies se1 = se2\big) \tag{5.9}$$

Axiom a-not–in-se-dif-a: *A generic element is not in the difference of any set with $singl(a)$*
Here we present the formula:

$$\forall * \;\; in(a, se) \implies (\neg\, in(a, diff\,(se, singl(a)))) \tag{5.10}$$

axiom	disjoint	locks	nexts	order	preserve	region
union-def	0	1	223	0	1	15
union-conmutative	0	1	2	0	0	1
a–in–singl-a	0	1	778	16	0	71
emptySet-is-empty	0	1	139	0	0	2
set-exten-inv	0	1	136	0	0	0
a-not–in-se-dif-a	0	1	47	0	0	12

Table 5.3: Uses of the set subset of axioms.

Memory

These set of axioms model Σ_{mem}, i.e. the axioms related with the updates of a heap memory mem. Table 5.1 reports the uses of each axiom when proving each invariant.

Axiom rd-mem–def: *The* cell *returned when reading the position null of any memory m is the error* cell.
Here we present the formula:

$$\forall * \quad rd(m, null) = error \tag{5.11}$$

Axiom upd–def–not-null: *(Update definition)*
Let a be an addr *different from null, m a* mem *and c a* cell. *We call m2 the result of the upd statement. Then, updating the value of m stored at address a with a* cell *c implies that c is the value returned when reading in the resulting memory the stored at a.*
Here we present the formula:

$$\forall * \quad ((\neg\, a = null) \implies upd(m, a, c) = m2 \implies rd(m2, a) = c) \tag{5.12}$$

Axiom upd–def–one-at-the-time: *Let a be an* addr *different from null, m a memory. We call m2 the result of the upd statement. An upd statement that only modifies the cell at a, preserves all other values.*
Here we present the formula:

$$\forall * \quad (((\neg\, a = null) \wedge (\neg\, a = b)) \implies upd(m, a, c) = m2 \implies rd(m, b) = rd(m2, b)) \tag{5.13}$$

axiom	disjoint	locks	nexts	order	preserve	region
rd-mem–def	0	1	194	1729	1	8
upd–def–not-null	0	1	283	1729	17	6
upd–def–one-at-the-time	0	1	451	1803	19	10

Table 5.4: Uses of the subset of update axioms.

Element Arithmetic

This set of axiom models Σ_{elem}, i.e., the axioms related with the arithmetic of the elem. Table 5.1 reports the uses of each axiom when proving each invariant.

Axiom less-trans: *Transitivity of the order relation on* elems.
Here we present the formula:

$$\forall * \quad ((x < y \wedge y < z) \implies x < z) \tag{5.14}$$

Axiom ls-xy–not-ls-yx: *The order relation* elems *is total.*
Here we present the formula:

$$\forall * \quad (ls\_elem(x, y) \iff ((\neg\, x = y) \wedge (\neg\, ls\_elem(y, x)))) \tag{5.15}$$

Axiom lowest–less-than-highest: *The lowestElem is less than the highestElem (according to the order relation defined among elements).*
Here we present the formula:

$$\forall * \quad ls\_elem(lowestElem, highestElem) \tag{5.16}$$

axiom	disjoint	locks	nexts	order	preserve	region
less-trans	0	1	0	1271	0	0
ls-xy–not-ls-yx	0	1	6	1513	0	0
lowest–less-than-highest	0	1	0	1535	0	0
lowestElem–def-tll	0	1	13	1164	0	0
highestElem–def-tll	0	1	6	1554	0	0

Table 5.5: Uses of the subset of the element arithmetic axioms.

Axiom lowestElem–def-tll: *(Definition of lowestElem)*
Any other element is great or equal than the lowestElem.
Here we present the formula:

$$\forall * \quad (e = lowestElem \vee lowestElem < e) \tag{5.17}$$

Axiom highestElem–def-tll: *(Definition of highestElem)*
Any other element is less or equal than the highestElem.
Here we present the formula:

$$\forall * \quad (e = highestElem \vee e < highestElem) \tag{5.18}$$

Addr2set

This subset of axioms and the following are specific axioms introduced this thesis. These are more complex than the previous axioms because they have the substance of the theories of liked list in the heap.

Axiom nextreg: *Let a be an* addr *reachable from another* addr *b. If a is different from null, then the node pointed by a is also reachable from b.*
Here we present the formula:

$$\forall * \quad ((in(a, se) \wedge se = addr2set(m, b) \wedge c = next(rd(m, a)) \wedge (\neg\, a = null)) \implies in(c, se)) \tag{5.19}$$

Axiom lock-keeps-addr2set: *The set of* addr *reachable from an* addr *hd is preserved after a lock statement targeting another (or the same)* addr *a. This is true because a lock statement does not change connectivity properties but only the thread using the* addr*. Even if the lock statement targets error, the reachable* addr *are preserved.*
Here we present the formula:

$$\begin{aligned} \forall * \quad & (hp_p = upd(hp, a, mkcell(data(rd(hp, a)), next(rd(hp, a)), t))) \\ & \implies (addr2set(p, hd) = addr2set(hp_p, hd)) \end{aligned} \tag{5.20}$$

Axiom addr2set-null-is-singl-null: *The set of* addr *reachable from null is the* setaddr *with just the* addr *null. This is true as a consequence of previous axioms, because the next of error is null for every memory.*
Here we present the formula:

$$\forall * \quad addr2set(m, null) = singl(null) \tag{5.21}$$

axiom	disjoint	locks	nexts	order	preserve	region
nextreg	0	1	112	0	1	68
lock-keeps-addr2set	0	0	0	0	17	0

Table 5.6: Uses of the subset of axioms related with Σ_{path}, i.e., the reachability of the addr.

Important

Axiom lock-keeps-heap–data & lock-keeps-heap–next: *In the same way a lock statement preserves the* setaddr *reachable from a given* addr*, the same principle applies to some other functions apart from addr2set. A lock statement does not change the data and next values of a* cell*.*
Here we present the formula:

$$\begin{aligned} &\forall * \quad (hp_p = upd(hp, a, mkcell(data(rd(hp, a)), next(rd(hp, a)), t))) \\ \implies &(data(r(hp, a)) = data(rd(hp_p, a)) \wedge next(r(hp, a)) = next(rd(hp_p, a))) \end{aligned} \tag{5.22}$$

Axiom addr2set-rec-def: *Recursive definition of addr2set. As the addr2set is used to get the reachable* addr *from another* addr *a, it is to expect that the addr2set of the next of a would be the same* setaddr *excluding a. This definition additionally states that an* addr *is reachable from itself.*
Here we present the formula:

$$\forall * \quad addr2set(m, a) = union(singl(a), addr2set(m, next(rd(m, a)))) \tag{5.23}$$

Axiom addr2set-primim: *This axiom is the base of the recursion in the definition of addr2set.*
Here we present the formula:

$$\begin{aligned} \forall * \quad &((data(rd(hp, hd)) < data(rd(hp, tl)) \wedge next(rd(hp, hd)) = tl \\ &\wedge next(r(hp, tl)) = null) \implies addr2set(hp, hd) = union(union(hd, tl), singl(null))) \end{aligned} \tag{5.24}$$

Axiom not-in-region–not-change-heap-addr: *Let hd be an* addr *and hp a* mem*. Then, modifying a* mem *in an* addr *which is not reachable from hd preserves the* setaddr *of reachable* addr *from hd.*
Here we present the formula:

$$\forall * \quad ((\neg\, in(a, addr2set(hp, hd))) \implies addr2set(hp, hd) = addr2set(upd(hp, a, c), hd)) \tag{5.25}$$

Axiom insert–keeps-addr2set: *This axiom allows to upd a* mem *preserving the addr2set. This upd correspond to an insertion in the list.*
Let hp be a mem*. Let hd be an* addr*. We call reg the set of address reachable from hd as captured by addr2set. Let prev be an* addr *reachable from hd (in the addr2set of hp, hd) and curr, aux two other* addr*. Let the three addresses (prev, curr, aux) be different from one another and all of them different from null.*
If prev points curr and aux points curr, then making prev point to aux has the following effect on

reg: aux now is in reg and every reachable address before the upd is still reachable.
Here we present the formula:

$$\begin{aligned}&\forall * \quad (reg = addr2set(hp, hd) \wedge union(reg, singl(aux)) = reg_p \wedge \\ &next(rd(hp, prev)) = curr \wedge (\neg\, prev = curr) \wedge next(rd(hp, aux)) = curr \wedge (\neg\, aux = null) \wedge \\ &(\neg\, prev = null) \wedge (\neg\, curr = null) \wedge in(prev, addr2set(hp, hd)) \wedge \\ &hp_p = upd(hp, prev, mkcell(data(rd(hp, prev)), aux, rd(hp, prev).lockid))) \\ &\quad \Longrightarrow\ reg_p = addr2set(hp_p, hd)\end{aligned} \tag{5.26}$$

Axiom remove–keeps-addr2set: *In a similar way to (5.1), this axioms allows to upd a* mem *preserving the addr2set, but this time corresponding to a removal in the list.*
Let hp be a mem*. Let hd be an* addr*. We call reg the set of addresses reachable from hd. Let prev be reachable from hd and curr, aux two other* addr*, such that the three address (prev, curr, aux) are different from each other and all of them are not null. If prev points to curr, aux points to curr, then making prev point to aux has the following effect in the reg: curr is now not reachable but every* addr *reachable before the upd is still reachable.*
Here we present the formula:

$$\begin{aligned}&\forall * \quad ((next(rd(hp, curr)) = aux \wedge next(rd(hp, prev)) = curr \wedge (\neg\, aux = next(rd(hp, prev))) \\ &\wedge hp_p = upd(hp, prev, mkcell(data(rd(hp, prev)), aux, rd(hp, prev).lockid)) \\ &\wedge (\neg\, aux = null) \wedge in(curr, addr2set(hp, hd)) \wedge in(null, addr2set(hp, hd)) \\ &\wedge in(prev, addr2set(hp, hd))) \implies diff(addr2set(hp, hd), singl(curr)) = addr2set(hp_p, hd))\end{aligned} \tag{5.27}$$

Axiom order-primim: *This axiom is the base of the recursion in the definition of orderlist. Let hp be a* mem *and hd, tl be two* addr*. If the* elem *stored in hp at hd is less than the* elem *stored in hp at tl and hd point to tl, then the list from hd to tl is ordered.*
Here we present the formula:

$$\begin{aligned}&\forall * \quad (ls\_elem(data(rd(hp, hd)), data(rd(hp, tl))) \wedge tl = next(rd(hp, hd)) \\ &\quad \wedge next(rd(hp, tl)) = null) \implies orderlist(hp, hd, tl)\end{aligned} \tag{5.28}$$

Axiom insert–keeps-orderlist: *This axioms allows to update a* mem *preserving the orderlist predicate. This update preserves the predicate if certain conditions are satisfied.*
Let hp be a mem *and hd, tl be two* addr *which satisfy orderlist(hp, hd, tl); let tl be different from null, but pointing to null. Let prev, aux, curr be* addr *with ordered elements, i.e. the* elem *stored in hp at prev is less than the one at aux, which is less than the one at curr. In addition, curr is pointed by aux and prev.* ***Then****, updating hp so that prev points to aux preserve orderlist* (hp', hd, tl)*.*
Here we present the formula:

$$\begin{aligned}&\forall * \quad ((orderlist(hp, hd, tl) \wedge (\neg\, tl = null) \wedge next(rd(hp, tl)) = null \\ &\wedge data(rd(hp, prev)) < data(rd(hp, aux)) \wedge data(rd(hp, aux)) < data(rd(hp, curr)) \\ &\wedge next(rd(hp, aux)) = curr \wedge next(rd(hp, prev)) = curr \\ &\wedge hp_p = upd(hp, prev, mkcell(data(rd(hp, prev)), aux, rd(hp, prev).lockid))) \implies orderlist(hp_p, hd, tl))\end{aligned} \tag{5.29}$$

axiom	disjoint	locks	nexts	order	preserve	region
lock-keeps-heap–data	0	0	0	98	0	0
lock-keeps-heap–next	0	1	2749	0	0	1
addr2set-rec-def	0	1	63	0	1	4
addr2set-primin	0	0	0	0	1	0
not-in-region–not-change-heap-addr	0	1	7	16	2	0
insert–keeps-addr2set	0	1	52	0	0	53
remove–keeps-addr2set	0	1	0	0	0	6
order-primin	0	0	0	0	1	0
insert–keeps-orderlist	0	1	5	0	0	5
remove–keeps-orderlist	0	1	5	0	0	5
next-is-not-same–if-ordered	0	1	7	0	0	0

Table 5.7: Uses of the important subset of axioms

Axiom remove–keeps-orderlist: *In a similar way to (5.1), this axiom allows to update a* mem *preserving the orderlist predicate, but this update corresponds to a removal in the list. This update preserves the predicate if certain conditions are satisfied.*
Let hp be a mem *and hd, tl be two* addr *which satisfy orderlist*(hp, hd, tl) *and that tl points to null. Let* $prev, aux, curr$ *satisfy that aux is pointed by curr, which is pointed by prev. In addition,* $curr, prev, aux$ *must be reachable from hd.* ***Then****, updating hp so that hp stores at prev the same cell stored at aux is an action that preserves orderlist for the new hp from hd to tl.*
Here we present the formula:

$$\begin{aligned}\forall * \quad & ((aux = next(rd(hp, curr)) \wedge curr = next(rd(hp, prev)) \\ & \wedge (\neg\, aux = null) \wedge null = next(rd(hp, tl)) \wedge (\neg\, aux = next(rd(hp, prev))) \\ & \wedge hp_p = upd(hp, prev, mkcell(data(rd(hp, prev)), aux, rd(hp, prev).lockid)) \\ & \wedge in(prev, addr2set(hp, hd)) \wedge in(curr, addr2set(hp, hd)) \\ & \wedge in(null, addr2set(hp, hd)) \wedge in(aux, addr2set(hp, hd)) \\ & \wedge orderlist(hp, hd, tl)) \implies orderlist(hp_p, hd, tl))\end{aligned} \tag{5.30}$$

Axiom next-is-not-same–if-ordered: *Let* hd, tl *and a be* addr *reachable from hd in the* mem *hp, with* hp, tl *and a different from null. In addition, let tl point to null.* ***Then*** *a can not point to itselt.*
Here we present the formula:

$$\begin{aligned}\forall * \quad & ((in(a, addr2set(hp, hd)) \wedge in(tl, addr2set(hp, hd)) \wedge (\neg\, hd = null) \\ & \wedge (\neg\, tl = null) \wedge (\neg\, a = null) \wedge next(rd(hp, tl)) = null) \implies (\neg\, next(rd(hp, a)) = a))\end{aligned} \tag{5.31}$$

The full list of axioms has been listed in appendix B.

We claim that there is no need for more axioms than the ones defined at 5.1. The next section will present which axioms are needed for proving each invariants.

5.2 Analysis of generated proofs

Example Lets take disj[2] as an example. We aim to prove that disj is an invariant. To do so, we need to prove its VCs valid. There are only needed 3 axioms: the axiom which states that numbers are different and the 2 axioms which states i and j are threads. With these 3 axiom, Spass can prove valid all the Spass problems associated with disj.

To clarify the process described in 4.1.1 we expose all the Spass problems generated to prove that disj is an invariant. There are 362 Spass problems. Half of them correspond to the *Reduced Spass problem* and the other half to the *Reduced implies original* problem. For each half, there are 181 Spass problems: For **initiation** there is one problem. For self-consecution there are 60 Spass problems. The program has 55 statements, so there is a VC for each line except for *while* and *if* statements which have 2 VC associated depending on the validity of the condition. As there are 3 while and 2 if, we have 60 Spass problems for each thread. As disj involves 2 threads, there are **120 self-consecution Spass problems**. Finally for **Others consecution** there are **60** Spass problems.

Analysis Most of the transitions (180 in 5178) are proven without any relevant axiom. Some of them do not even need any axiom. *Reduced implies original* problems usually do not need any axiom, because there are simple but big FOL formulas.

Table 5.2 contains some global information about the proof of each invariant. It is important to mention that every invariant has 2 problems of initiation.

Invariant	Self-consecution	Others consecution	Total	Number of axioms used
disjoint	240	120	362	3
locks	120	1254	1376	79
nexts	120	1254	1376	74
order	120	1308	1430	55
region	120	174	296	63
preserve	0	336	338	49

Table 5.8: Number of Spass problems by invariant.

In terms of the number of axioms needed, it is interesting to see that lock is the most complicated, followed by next. The easiest is disj, which only uses *nums-are-different* (5.1) and axioms related with sorts (there are 2 threads involved in the formula which are different from one another.)
The differences on the number of other-consecution problem is caused by the splitting on complex Spass problems. To prove order all other-consecution problems have been splat, but for lock and next just 21 other consecution problems have been splat. Despite that fact, order is not very complex in terms of total number of axioms needed.

Another interesting data to look at in order to study the difficult of each invariant is the number of axioms needed in each problem. Figure 5.1 shows the number of problems solved against the number of axioms needed. The graphic illustrate the number of problems solved normalized by the total amount of problems that invariant involves. As it was stated before, most of the problems can be proven without any axiom, or just with the axioms of sorts and

[2]Its full definition is at D

numbers to reason about the pc. Looking to the figure, the difficulty of each axiom in terms of the axioms used in each problem is more precise: order needs more axioms in general than the rest.

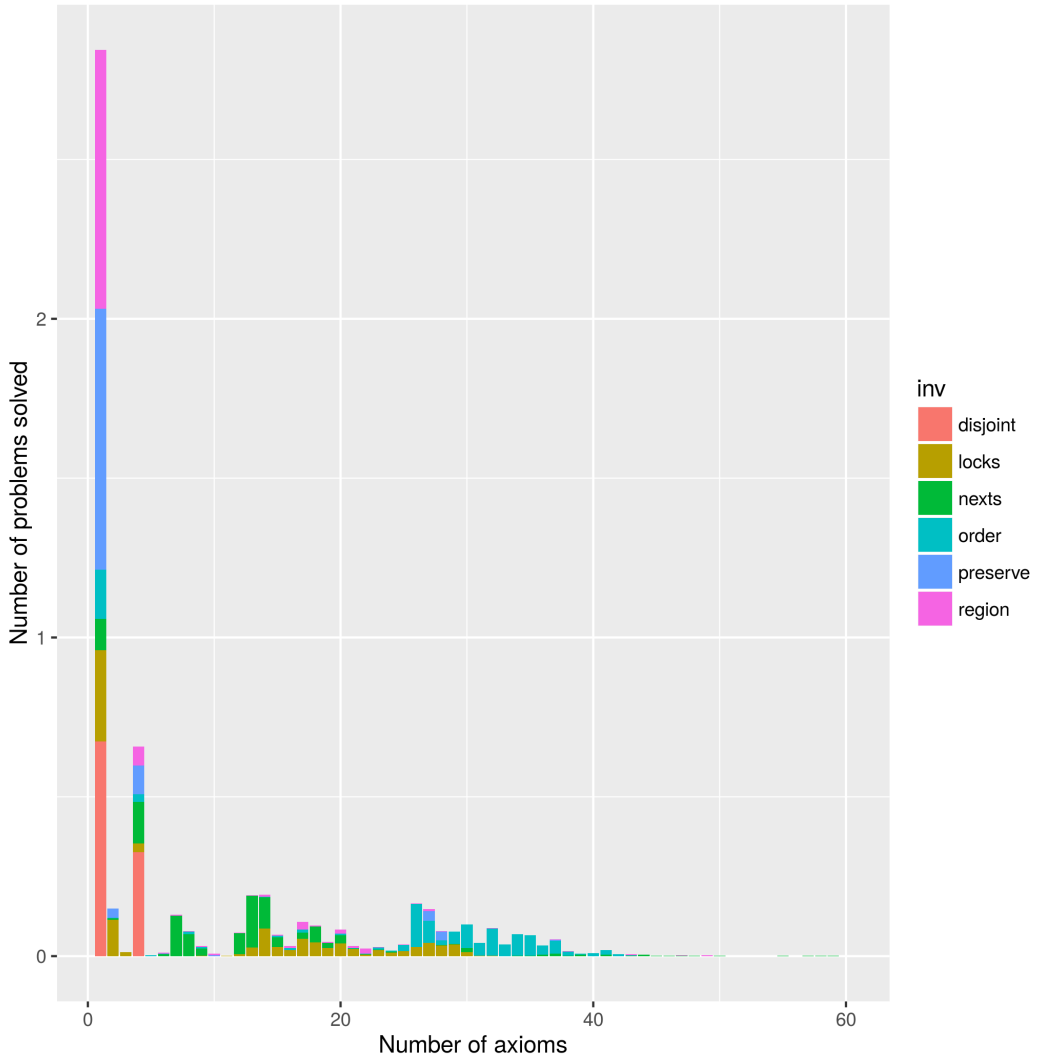


Figure 5.1: Number of problems solved against number of axiom needed.

5.2.1 Special Transitions

Due to the limited space, we can not explore deeply the proof of the invariants. However, some we offer a brief analysis of the complicated transitions for the most important invariant: list. The complicated transitions are the ones modifying the heap: *locks*, *insertion* and *removal*. The transitions including *locks* and *unlocks* can be proven easily with the axioms, because a *lock* does not modify the content nor the pointer; thus, it can not break the list nor disorder the list. For the other transitions some expert knowledge is needed, because Spass lacks information to finish the proof. This limitation is that Spass does not know how to choose the axioms to use, leading into an unnecessary exploration of a branch of the proof. If Spass could know that, it would be much more efficient. As a consequence, transitions 35 and 55 have been subdivided in to a number of smaller Spass problems, reducing manually in each case the number of axioms needed

and leaving the ones which could be relevant.

5.3 Time analysis

5.3.1 Proof generation

All the times shown refer to the time Spass takes to prove the validity of each problem. Previous parsing and creating the proper input to Spass has been ignored.

5.3.2 Proof Checking

One of the greatest consequences of this method is the possibility to refute the generated proofs and double check that they are valid. We developed the solution such that for each Spass problem there are three files generated. The Spass output (storing the information of which axioms have been used and the running time), the Spass proof in Spass format and the Spass proof in TPTP syntax ([21]). By generating a TPTP proof for each Spass problem we made possible to check these proofs with any other theorem prover as "The Vampire and the FOOL" [10] or *Isabelle: A generic theorem prover* [11].

Another advantage of the generation of these files is the reduced amount of time needed to check the validity of the fine-grained-linked-list implementation. There is no need to regenerate all the verification conditions and regenerate all the Spass files and let Spass run until it finds the proof for every problem. One can just run Spass on the generated proofs and will save lot of time. Lets see the amount of time needed to check the proofs instead of generate them.

5.3.3 Comparing times

In order to use formal verification in real environments, a time analysis is fundamental. The more time it takes, the more useless formal verification is. How much time does it takes to generate all the proofs for this linked-list implementation? And more importantly, How much time does it takes to check the generated proofs?

In Table 5.9 one can find the summary of the timing according to four different activities. The first column refers to the time that Leap takes to prove the invariant (obtained from [13]). The second column, *Full process*, refers to the time the whole process takes[3]. This is, generating the verification conditions, creating all the Spass problems and letting Spass work to solve each problem. The third column, *Sum of Spass time*, refers to the amount of time that Spass takes to solve all the problems of each invariant. This column shows the real time necessary to generate the proofs, from the already generated files. Finally, the last column shows very important information, which is amount of time needed to check that the proofs are correct is less than every other method. This method reduces significantly the time spent in checking the formal verification of a problem.

There are some very relevant issues with the measures of the *Full process* time and the *Sum of Spass time*.

[3]The full process is described in 4.1.1

Invariant	Leap	Full process	Sum of Spass time	Check proofs
list	12 min 85 sec	∞	10 min 56.56 sec	0 min 52.19 sec
order	1 min 20 sec	180 min 40.548 sec	62 min 32.89 sec	7 min 41.17 sec
lock	0 min 50 sec	39 min 28.321 sec	13 min 34.00 sec	1 min 53.94 sec
next	1 min 76 sec	352 min 17.839 sec	123 min 37.73 sec	17 min 06.12 sec
region	25 min 67 sec	7 min 34.425 sec	1 min 14.68 sec	0 min 19.14 sec
disj	0 min 22 sec	4 min 13.300 sec	1 min 17.15 sec	0 min 26.40 sec

Table 5.9: Compare of the times.

About measuring the full process In order to improve performance, all Spass problems are first considered as if they were the simplest. If it took Spass too much time to (according to a timeout) to find a proof for a problem, then the problem would be automatically divided. This approach allows to prove simple problems very quickly, such as the majority of disj transitions for example. When a problem is not so simple (again, according to the timeout), then it is divided and some time has been wasted because the initial attempt to prove the problem is terminated. The value of this timeout is not trivial to be set. There are transitions which do not need any axioms. This is caused because of VCs of the form ($false \rightarrow something$). For these transitions, it is usual that Spass takes less than one second to prove them. Thus, we could set the timeout to one second. If a transition is taking more than one second, then it may need more axioms and or it should be divided. But we may incur into dividing a problem which does not need to be divided. Others consecution for transition 33 in lock takes 1 second and a half, but does not need to be divided because it is of the form ($false \rightarrow something$). A timeout of 1 second would make Spolv to divided this transition, but a timeout of 2 seconds would cause a waste of 1 second in each not simple Spass problem. The timeout had been setted to 100 seconds because it was preferable in order to generate less Spass problems, and consequently less proofs.

The problem explained before acquires more relevance in another context. Spass does not have an expert knowledge about which axioms should be used first because they are relevant and which should not. No arithmetic axiom is needed to prove region and Spass does not neither need any axiom about the order relation between elements. There is a subset of axioms for each invariant. Including unnecessary axioms may cause Spass to take more time searching for the proof in a branch where the proof cannot be found. It may not cause any delay because it finds the proof without exploring all the branches. We show an example of this reality. Transition 11 of the list takes 23 seconds to Spass when all the axioms are included but it takes 0.44 seconds when only used axioms are included. This difference has been an important challenge, because when Spass could not find a proof it did not necessarily mean that the problem was unsatisfiable. It could mean that too many unnecessary axiom had been included or that the problem needed to be divided. Thus, the third column shows the real amount of time that Spass has needed to generate the proofs. This time is not the lowest bound one could get, but as one can see, it is less than the whole process. For the transition mentioned before, Spass takes 0.11 seconds to check that the proof is correct. For list there are two transitions whose proof were completed manually, as we explained in 5.2.1.

6 Conclusions

Abstract We will conclude the thesis with an abstract of the information we discovered about the linked list theory. Finally, we will discuss the scalability to production of this way of verifying programs. Is this work worth it in every case? The advantages and disadvantages will be discussed.

6.1 Linked list is valid

The main conclusion of this bachelor thesis is that the implementation of a fine grained lock coupling linked list shown at 3.3.1 is valid and has been verified. We can claim that that implementation always preserves the order of the stored elements and its structure of a list regardless the number of threads executing. There is no need of testing and it cannot fail because some possible scenario was not tested. If we trust the axioms in which the proofs are based (which are not difficult to understand and trust because of their simplicity), we must accept the conclusion that the order and list structure are preserved.

6.1.1 Reproducible proofs

Additionally, the generated proofs of that conclusion have been stored so any one can reproduce the proofs and double-check them with another theorem prover.

6.2 Discussion about formal verification

The amount of work needed to obtain the proofs of the validity of the program may seem too much. A whole bachelor thesis to prove something not very difficult to see looking carefully at the code and testing some of examples.

There is a fact which should not be forgotten, before we discuss about formal verification. This work has been realized to explore the possibility of complementing Leap with first order proofs, to cover some lacks of Leap[1]. Additionally, this work has been done from the basis of formalism, using a theorem prover. We have seen along the thesis that Spass has some difficulties to select the best branch to explore during the proofs, and we fulfill this problem trying to reduce the number of axioms Spass could use, leading to try to prove twice the same problem. If another more specific prover were used perhaps the proofs would have been found easier and rapidly. Additionally, if the goal was to develop a verified tool, we would have chosen another path. We could have used VCC and a C implementation of a fine-grained lock coupling linked list in order to put the implementation into production and it would have been a lot easier, but the goal was not just prove the implementation but research formal verification from the very begging of the process.

Despite that fact, the certainty we have that this linked list is valid can not be obtained by testing. If one accept the axioms, then the validity must be accepted too with absolute certainty.

6.3 Increasing Leap Performance

There are some difficult transitions in which Leap spend significantly more time than in others. We thought that may be, including some axioms in the decision procedure of Leap would increase Leap performance. Some preliminary analysis showed us that including axioms in difficult transitions would not made any performance difference so we did not explore deeply that possibility.

[1]The main lack is the impossibility to reproduce verification without rerunning all the test

Bibliography

[1] Alejandro Sánchez. "Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures". PhD thesis. Universidad Politécina de Madrid, 2015.

[2] Weidenbach C., Dimova D., Fietzke A., Kumar R., Suda M., and Wischnewski P. *SPASS Version 3.5*. LNCS 5663. 22nd International Conference on Automated Deduction CADE 2009.

[3] *Software update destroys $286 million Japanese satellite*. hackaday.com.

[4] *Boeing 787 bug could cause loss of control*. theguardian.com.

[5] *Computer-controlled machine overdosed six people*. sunnyday.mit.edu.

[6] Leopold Löwenheim. *A Source Book in Mathematical Logic*. Harvard Univ. Press. 1915.

[7] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. *Finding and understanding bugs in C compilers*. 2011.

[8] Zaynah Dargaye. "Vérification formelle d'un compilateur pour langages fonctionnels". PhD thesis. Université Paris 7 Diderot, 2009.

[9] Cohen Ernie, Dahlweid Markus, Hillebrand Mark, Leinenbach Dirk, Moskal Michał, Santen Thomas, Schulte Wolfram, and Tobies Stephan. *A Practical System for Verifying Concurrent C*. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009). Provides a good overall system description of VCC.

[10] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. "The Vampire and the FOOL". In: *CoRR* abs/1510.04821 (2015). URL: `http://arxiv.org/abs/1510.04821`.

[11] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994.

[12] Alejandro Sánchez. *Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures*. 2015.

[13] Alejandro Sánchez and César Sánchez. "Parametrized Invariance for Infinite State Processes". In: *CoRR* abs/1312.4043 (2013).

[14] Alejandro Sánchez and César Sánchez. "A Theory of Skiplists with Applications to the Verification of Concurrent Datatypes". In: LNCS 6617 (2011), pp. 343–358.

[15] Christoph Weidenbach. "Spass input syntax version 1.5". In: *Max-Planck-Institut fur Informatik* (2008).

[16] Jason Hickey Yaron Minsky Anil Madhavapeddy. *Real World OCAML*. O'Reilly Media, November 2013.

[17] *Ocamllex*. Chapter 16 in *Real World OCAML* (Yaron Minsky, November 2013) [16].

[18] *Ocamlyacc*. `http://courses.softlab.ntua.gr/compilers/2015a/ocamlyacc-tutorial.pdf`.

[19] Alejandro Sánchez and César Sánchez. “Decision Procedures for the Temporal Verification of Concurrent Lists”. In: LNCS 6447 (2010), pp. 74–89.

[20] Bradley Mana. *The Calculus of Computation*. Springer, 2007.

[21] Geoff Sutcliffe and Christian Suttner. *Thousands of Problems for Theorem Provers*. www.tptp.org.

Index

Appendices

A

Inductive Assertion Method

This example is proposed to illustrate the inductive assertion method. A simple program, which one can see it should work perfectly is verified manually applying the method.

Example: *We study the loop version of the factorial function.*

l_1 :	$\mathtt{x} := 10$
l_2 :	$\mathtt{f} := 1$
l_3 :	***while*** $(\mathtt{x} \geq 1)$ ***do***
l_4 :	$\mathtt{f} = \mathtt{f} * \mathtt{x}$
l_5 :	$\mathtt{x} = \mathtt{x} - 1$
l_6 :	***end while***
l_7 :	$\cdots$

We seek to prove two formulae.

$$\varphi_1 \equiv (l_5 \vee l_4 \rightarrow \mathtt{x} \geq 1) \quad \wedge \quad \varphi_2 \equiv \mathtt{x} \geq 0$$

First, we reduce the program to its VC*.*

$$\begin{aligned}
\tau_1 &\equiv pc(T) = l_1 \wedge pc\prime(T) = l_2 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x\prime} = 10 \\
\tau_2 &\equiv pc(T) = l_2 \wedge pc\prime(T) = l_3 \wedge \mathtt{f\prime} = 1 \wedge x\prime = x \\
\tau_3 &\equiv pc(T) = l_3 \wedge pc\prime(T) = l_4 \wedge \mathtt{f\prime} = \mathtt{f} \wedge x\prime \geq 1 \\
\tau_4 &\equiv pc(T) = l_4 \wedge pc\prime(T) = l_5 \wedge \mathtt{f\prime} = \mathtt{f} * \mathtt{x} \wedge \mathtt{x\prime} = \mathtt{x} \\
\tau_5 &\equiv pc(T) = l_5 \wedge pc\prime(T) = l_3 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x\prime} = \mathtt{x} - 1 \\
\tau_6 &\equiv pc(T) = l_3 \wedge pc\prime(T) = l_7 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x} < 1
\end{aligned}$$

We need to prove, for $i = 1, 2$*:*

$$\begin{cases} \tau_1 \wedge \varphi_i \rightarrow \varphi_i\prime \\ \tau_2 \wedge \varphi_i \rightarrow \varphi_i\prime \\ \tau_3 \wedge \varphi_i \rightarrow \varphi_i\prime \\ \tau_4 \wedge \varphi_i \rightarrow \varphi_i\prime \\ \tau_5 \wedge \varphi_i \rightarrow \varphi_i\prime \\ \tau_6 \wedge \varphi_i \rightarrow \varphi_i\prime \end{cases}$$

$$\varphi_1$$

$\tau_1 \wedge \varphi_1 \rightarrow \varphi_1\prime$:

$$(\underbrace{pc(T) = l_1 \wedge pc\prime(T) = l_2 \wedge \mathtt{f}\prime = \mathtt{f} \wedge \mathtt{x}\prime = 10}_{\tau_1} \wedge (\underbrace{[pc(T) = l_5 \vee pc(T) = l_4] \rightarrow \mathtt{x} \geq 1}_{\varphi_1})) \rightarrow (\underbrace{[pc\prime(T) = l_5 \vee pc\prime(T) = l_4] \rightarrow \mathtt{x}\prime \geq 1}_{\varphi_1\prime})$$

The formula is valid because $pc(T) = l_2 \neq l_5 \wedge l_2 \neq l_4$ *thus the* $\varphi\prime_1$ *is true.*

$\tau_2 \wedge \varphi_1 \rightarrow \varphi_1\prime$:

$$(\underbrace{pc(T) = l_2 \wedge pc\prime(T) = l_3 \wedge \mathtt{f}\prime = 1 \wedge x\prime = x}_{\tau_2} \wedge (\underbrace{[pc(T) = l_5 \vee pc(T) = l_4] \rightarrow \mathtt{x} \geq 1}_{\varphi_1})) \rightarrow (\underbrace{[pc\prime(T) = l_5 \vee pc\prime(T) = l_4] \rightarrow \mathtt{x}\prime \geq 1}_{\varphi_1\prime})$$

The formula is valid because $pc(T) = l_3 \neq l_5 \wedge l_2 \neq l_4$ *thus the* $\varphi\prime_1$ *is true.*

$\tau_3 \wedge \varphi_1 \rightarrow \varphi_1\prime$:

$$(\underbrace{pc(T) = l_3 \wedge pc\prime(T) = l_4 \wedge \mathtt{f}\prime = \mathtt{f} \wedge x\prime \geq 1}_{\tau_3} \wedge (\underbrace{[pc(T) = l_5 \vee pc(T) = l_4] \rightarrow \mathtt{x} \geq 1}_{\varphi_1})) \rightarrow (\underbrace{[pc\prime(T) = l_5 \vee pc\prime(T) = l_4] \rightarrow \mathtt{x}\prime \geq 1}_{\varphi_1\prime})$$

The formula is valid because $x\prime \geq 1$ *thus the* $\varphi\prime_1$ *is true.*

$\tau_4 \wedge \varphi_1 \rightarrow \varphi_1\prime$:

$$(\underbrace{pc(T) = l_4 \wedge pc\prime(T) = l_5 \wedge \mathtt{f}\prime = \mathtt{f} * \mathtt{x} \wedge \mathtt{x}\prime = \mathtt{x}}_{\tau_4} \wedge (\underbrace{[pc(T) = l_5 \vee pc(T) = l_4] \rightarrow \mathtt{x} \geq 1}_{\varphi_1})) \rightarrow (\underbrace{[pc\prime(T) = l_5 \vee pc\prime(T) = l_4] \rightarrow \mathtt{x}\prime \geq 1}_{\varphi_1\prime})$$

The formula is equivalent (applying resolution) to

$$(\mathtt{x\prime} = \mathtt{x} \wedge \mathtt{x} \geq 1) \rightarrow (\mathtt{x\prime} \geq 1)$$

Which is valid because of equality congruence.

$\tau_5 \wedge \varphi_1 \rightarrow \varphi_1\prime$*:*

$$(\underbrace{[pc(T) = l_5 \vee pc(T) = l_4] \wedge pc\prime(T) = l_3 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x\prime} = \mathtt{x} - \mathtt{1}}_{\tau_5} \wedge (\underbrace{[pc(T) = l_5 \vee pc(T) = l_4] \rightarrow \mathtt{x} \geq 1}_{\varphi_1}))$$
$$\rightarrow (\underbrace{[pc\prime(T) = l_5 \vee pc\prime(T) = l_4] \rightarrow \mathtt{x\prime} \geq 1}_{\varphi_1\prime})$$

The formula is valid because $pc(T) = l_3 \neq l_5 \wedge l_2 \neq l_4$ *thus the* $\varphi\prime_1$ *is true.*

$\tau_6 \wedge \varphi_1 \rightarrow \varphi_1\prime$*:*

$$(\underbrace{pc(T) = l_3 \wedge pc\prime(T) = l_7 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x} < \mathtt{1}}_{\tau_6} \wedge \underbrace{pc(T) = l_5 \rightarrow \mathtt{x} \geq 1}_{\varphi_1})$$
$$\rightarrow (\underbrace{[pc\prime(T) = l_5 \vee pc\prime(T) = l_4] \rightarrow \mathtt{x\prime} \geq 1}_{\varphi_1\prime})$$

The formula is valid because $pc(T) = l_7 \neq l_5$ *thus the* $\varphi\prime_1$ *is true.*

Conclusion: *we have proven that* $pc(T) = l_5 \rightarrow x \geq 1$*. This is called an* ***invariant*** *because it is always true in all executions of the program. This invariant has been chosen specially because it is needed in the proof of* φ_2*.*

——————————— φ_2 ———————————

$\tau_1 \wedge \varphi_2 \rightarrow \varphi_2\prime$*:*

$$(\underbrace{pc(T) = l_1 \wedge pc\prime(T) = l_2 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x\prime} = 10}_{\tau_1} \wedge \underbrace{\mathtt{x} \geq 0}_{\varphi_2}) \rightarrow \underbrace{\mathtt{x\prime} \geq 0}_{\varphi_2\prime}$$

The formula is valid because $x\prime = 10 \rightarrow x\prime \geq 0$*.*

$\tau_2 \wedge \varphi_2 \rightarrow \varphi_2\prime$*:*

$$(\underbrace{pc(T) = l_2 \wedge pc\prime(T) = l_3 \wedge \mathtt{f\prime} = 1 \wedge x\prime = x}_{\tau_2} \wedge \underbrace{\mathtt{x} \geq 0}_{\varphi_2}) \rightarrow \underbrace{\mathtt{x\prime} \geq 0}_{\varphi_2\prime}$$

The formula is valid because of the congruence of equality used in $x\prime = x \wedge x \geq 0 \rightarrow x\prime \geq 0$

$\tau_3 \wedge \varphi_2 \rightarrow \varphi_2\prime$*:*

$$(\underbrace{pc(T) = l_3 \wedge pc\prime(T) = l_4 \wedge \mathtt{f\prime} = \mathtt{f} \wedge x\prime \geq 1}_{\tau_3} \wedge \underbrace{\mathtt{x} \geq 0}_{\varphi_2}) \rightarrow \underbrace{\mathtt{x\prime} \geq 0}_{\varphi_2\prime}$$

The formula is valid because $x\prime \geq 1 \rightarrow x\prime \geq 0$. $\tau_4 \wedge \varphi_2 \rightarrow \varphi_2\prime$*:*

$$(\underbrace{pc(T) = l_4 \wedge pc\prime(T) = l_5 \wedge \mathtt{f\prime} = \mathtt{f} * \mathtt{x} \wedge \mathtt{x\prime} = \mathtt{x}}_{\tau_4} \wedge \underbrace{\mathtt{x} \geq 0}_{\varphi_2}) \rightarrow \underbrace{\mathtt{x\prime} \geq 0}_{\varphi_2\prime}$$

The formula is valid because of the congruence of equality used in $x\prime = x \wedge x \geq 0 \rightarrow x\prime \geq 0$

$\tau_5 \wedge \varphi_2 \rightarrow \varphi_2\prime$*:*

$$(\underbrace{pc(T) = l_5 \wedge pc\prime(T) = l_3 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x\prime} = \mathtt{x} - \mathtt{1}}_{\tau_5} \wedge \underbrace{\mathtt{x} \geq 0}_{\varphi_2}) \rightarrow \underbrace{\mathtt{x\prime} \geq 0}_{\varphi_2\prime}$$

The formula has some more difficulty. Inside the loop x *should be greater than 1. However, that information is not encoded in the formula.*

The solution is use some ***support****. A support formula is an invariant formula added to the antecedent of an implication to give more information. This addition does not change the validity of the formula. We could equivalently prove*

$$(\tau_5 \wedge \varphi_1 \wedge \varphi_2 \rightarrow \varphi_2\prime) \rightarrow (\varphi_2 \rightarrow \varphi_2\prime)$$

And this is exactly the solution to proof this VC

$$(\underbrace{pc(T) = l_5 \wedge pc\prime(T) = l_3 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x\prime} = \mathtt{x} - \mathtt{1}}_{\tau_5} \wedge \underbrace{\mathtt{x} \geq 0}_{\varphi_2} \wedge \underbrace{[pc(T) = l_5 \vee pc(T) = l_4] \rightarrow \mathtt{x} \geq 1)}_{\varphi_1}) \rightarrow \underbrace{\mathtt{x\prime} \geq 0}_{\varphi_2\prime}$$

And this formula is valid. Applying resolution we get an equivalent valid formula:

$$(\mathtt{x\prime} = \mathtt{x} - \mathtt{1} \wedge \mathtt{x\prime} \geq 1) \rightarrow \mathtt{x} \geq 0$$

$\tau_6 \wedge \varphi_2 \rightarrow \varphi_2\prime$*:*

$$(\underbrace{pc(T) = l_3 \wedge pc\prime(T) = l_7 \wedge \mathtt{f\prime} = \mathtt{f} \wedge \mathtt{x} < \mathtt{1} \wedge \mathtt{x\prime} = \mathtt{x}}_{\tau_6} \wedge \underbrace{\mathtt{x} \geq 0}_{\varphi_2}) \rightarrow \underbrace{\mathtt{x\prime} \geq 0}_{\varphi_2\prime}$$

And this formula is valid. Applying resolution we get an equivalent valid formula:

$$(\mathtt{x\prime} = \mathtt{x} \wedge \mathtt{x} \geq 0 \rightarrow \mathtt{x\prime} \geq 0)$$

Conclusion *We have proof that* φ_1 *and* φ_2 *are invariants.*

B

Spass Syntax File and Full List of Axioms

```
begin_problem(Template).


list_of_descriptions.
 name({*Template*}).
 author({*Victor de Juan*}).
 status(unknown).
 description({*desc*}).
end_of_list.



list_of_symbols.
functions[

%% %% static global variables:
    region_prime,region,heap_prime,heap,elements_prime,elements,i,
%% %% local in threads and in procedures variables:
    search_prev_prime_i,search_prev_i,search_curr_prime_i,
       ↪ search_curr_i,search_aux_prime_i,search_aux_i,
       ↪ search_e_prime_i,search_e_i,search_e_prime_i,search_e_i,
       ↪ search_e_prime_i,search_e_i,remove_prev_prime_i,
       ↪ remove_prev_i,remove_curr_prime_i,remove_curr_i,
       ↪ remove_aux_prime_i,remove_aux_i,remove_e_prime_i,remove_e_i
       ↪ ,remove_e_prime_i,remove_e_i,remove_e_prime_i,remove_e_i,
       ↪ insert_prev_prime_i,insert_prev_i,insert_curr_prime_i,
       ↪ insert_curr_i,insert_aux_prime_i,insert_aux_i,
       ↪ insert_e_prime_i,insert_e_i,insert_e_prime_i,insert_e_i,
       ↪ insert_e_prime_i,insert_e_i,
```

```
%% %%   numbers:
     0,1,2,3,4,5,
%% %%  functions:

%%   %%  mem

(null,0),(upd,3),(rd,2),
%%   %%  W_reach


%%   %%  W_bridge

(path2set,1),(addr2set,2),(getp,3),(fstlock,2),
%%   %%  path

(epsilon,0),(consPath,1),
%%   %%  addr

(freshaddr,0),
%%   %%  setaddr

(emptySet,0),(union,2),(setDiff,2),(singl,1),
%%   %%  elem

(highestElem,0),(lowestElem,0),(main_e_prime_i,0),(main_e_i,0),
%%   %%  setelem

(emptySetElem,0),(unionElem,2),(setDiffElem,2),(singlElem,1),(
    ↪ set2elem,2),
%%   %%  cell

(error,0),(mkcell,3),(data,1),(next,1),(lockid,1),(lock,2),(unlock,1)
    ↪ ,(head,0),(tail,0),(freshcell,0),
%%   %%  nat

(s,1),
%%   %%  tid

(nothread,0),(pc_prime_i,0),(pc_i,0),
%%   %%  settid

(emptySetTh,0),(unionTh,2),(setDiffTh,2),(singlTh,1)
].


predicates[
 %  program
```

```
%%   %% mem

%%   %% W_reach
     (reach,4),
%%   %% W_bridge
     (orderlist,3),(initial,0),(search_result_prime_i,0),(
        ↪ search_result_i,0),
%%   %% path
     (append,3),
%%   %% addr

%%   %% setaddr
     (in,2),(sub,2),
%%   %% elem
     (ls_elem,2),
%%   %% setelem
     (inElem,2),(subElem,2),
%%   %% cell

%%   %% nat
     (ls,2),
%%   %% tid
     (ls_tid,2),
%%   %% settid
     (inTh,2),(subTh,2)
].
sorts[
     mem,path,addr,setaddr,elem,setelem,cell,nat,tid,settid
].
end_of_list.


list_of_formulae(axioms).
     %%%%%% sorts_types
     formula(tid(i),i__is__Tid_tllign). formula(equal(i,0),
        ↪ i__def_tllign).

     formula(setaddr(region_prime),setaddr_prime__def_tllign).
     formula(setaddr(region),setaddr__def_tllign).
     formula(mem(heap_prime),mem_prime__def_tllign).
     formula(mem(heap),mem__def_tllign).
     formula(setelem(elements_prime),setelem_prime__def_tllign).
     formula(setelem(elements),setelem__def_tllign).

     %%   %% mem
     formula(addr(null),null__is__addr_tllign).
     formula(forall([mem(h),addr(a),cell(c)],mem(upd(h,a,c))),
        ↪ upd__is__mem_tllign).
```

```
formula(forall([mem(h),addr(a)],cell(rd(h,a))),
    ↪ rd__is__cell_tllign).

%%   %% W_reach

%%   %% W_bridge
formula(forall([path(p)],setaddr(path2set(p))),
    ↪ path2set__is__setaddr_tllign).
formula(forall([mem(h),addr(a)],setaddr(addr2set(h,a))),
    ↪ addr2set__is__setaddr_tllign).
formula(forall([mem(h),addr(a),addr(a1)],path(getp(h,a,a1))),
    ↪ getp__is__path_tllign).
formula(forall([mem(h),path(p)],addr(fstlock(h,p))),
    ↪ fstlock__is__addr_tllign).

%%   %% path
formula(path(epsilon),epsilon__is__path_tllign).
formula(forall([addr(a)],path(consPath(a))),
    ↪ consPath__is__path_tllign).

%%   %% addr
formula(addr(freshaddr),freshaddr__is__addr_tllign).

%%   %% setaddr
formula(setaddr(emptySet),emptySet__is__setaddr_tllign).
formula(forall([setaddr(set_a),setaddr(set_a1)],setaddr(union(
    ↪ set_a,set_a1))),union__is__setaddr_tllign).
formula(forall([setaddr(set_a),setaddr(set_a1)],setaddr(setDiff(
    ↪ set_a,set_a1))),setDiff__is__setaddr_tllign).
formula(forall([addr(a)],setaddr(singl(a))),
    ↪ singl__is__setaddr_tllign).

%%   %% elem
formula(elem(highestElem),highestElem__is__elem_tllign).
formula(elem(lowestElem),lowestElem__is__elem_tllign).
formula(elem(main_e_prime_i),main_e_prime_i__is__elem_tllign).
formula(elem(main_e_i),main_e_i__is__elem_tllign).

%%   %% setelem
formula(setelem(emptySetElem),emptySetElem__is__setelem_tllign).
formula(forall([setelem(set_e),setelem(set_e1)],setelem(unionElem
    ↪ (set_e,set_e1))),unionElem__is__setelem_tllign).
formula(forall([setelem(set_e),setelem(set_e1)],setelem(
    ↪ setDiffElem(set_e,set_e1))),setDiffElem__is__setelem_tllign
    ↪ ).
formula(forall([elem(e)],setelem(singlElem(e))),
    ↪ singlElem__is__setelem_tllign).
formula(forall([setaddr(set_a),mem(h)],setelem(set2elem(set_a,h))
    ↪ ),set2elem__is__setelem_tllign).
```

```
%%   %% cell
formula(cell(error),error__is__cell_tllign).
formula(forall([elem(e),addr(a),tid(t)],cell(mkcell(e,a,t))),
    ↪ mkcell__is__cell_tllign).
formula(forall([cell(c)],elem(data(c))),data__is__elem_tllign).
formula(forall([cell(c)],addr(next(c))),next__is__addr_tllign).
formula(forall([cell(c)],tid(lockid(c))),lockid__is__tid_tllign).
formula(forall([cell(c),tid(t)],cell(lock(c,t))),
    ↪ lock__is__cell_tllign).
formula(forall([cell(c)],cell(unlock(c))),unlock__is__cell_tllign
    ↪ ).
formula(addr(head),head__is__addr_tllign).
formula(addr(tail),tail__is__addr_tllign).
formula(cell(freshcell),freshcell__is__cell_tllign).

%%   %% nat
formula(forall([nat(n)],nat(s(n))),s__is__nat_tllign).

%%   %% tid
formula(tid(nothread),nothread__is__tid_tllign).
formula(nat(pc_prime_i),pc_prime_i__is__nat_tllign).
formula(nat(pc_i),pc_i__is__nat_tllign).

%%   %% settid
formula(settid(emptySetTh),emptySetTh__is__settid_tllign).
formula(forall([settid(set_t),settid(set_t1)],settid(unionTh(
    ↪ set_t,set_t1))),unionTh__is__settid_tllign).
formula(forall([settid(set_t),settid(set_t1)],settid(setDiffTh(
    ↪ set_t,set_t1))),setDiffTh__is__settid_tllign).
formula(forall([tid(t)],settid(singlTh(t))),
    ↪ singlTh__is__settid_tllign).
formula(not(or(equal(i,nothread) )),th_nothread_diff_i_tllign).
% % % % % % % % % % Program axioms
% % % % % % % Natural axioms
% numbers:
formula(equal(s(0),1),def_1_tllign).
formula(equal(s(1),2),def_2_tllign).
formula(equal(s(2),3),def_3_tllign).
formula(equal(s(3),4),def_4_tllign).
formula(equal(s(4),5),def_5_tllign).
formula(and(not(equal(0,1)),not(equal(0,2)),not(equal(0,3)),not(
    ↪ equal(0,4)),not(equal(1,2)),not(equal(1,3)),not(equal(1,4))
    ↪ ,not(equal(2,3)),not(equal(2,4)),not(equal(3,4))),
    ↪ nums_are_different_tllign).

% < and s
formula(forall([nat(x),nat(y)],implies(equal(x,y),equal(s(x),s(y)
    ↪ ))),s_injective_tllign).
```

```
formula(forall([nat(x)],not(exists([nat(y)],equal(s(y),0)))),
    ↪ no_negative_numbers_tllign).
formula(addr(search_prev_prime_i),search_prev__is__addr_tllign).
    ↪     formula(addr(search_prev_i),search_prev__is__addr_tllign
    ↪ ).
formula(addr(search_curr_prime_i),search_curr__is__addr_tllign).
    ↪     formula(addr(search_curr_i),search_curr__is__addr_tllign
    ↪ ).
formula(addr(search_aux_prime_i),search_aux__is__addr_tllign).
    ↪ formula(addr(search_aux_i),search_aux__is__addr_tllign).
formula(elem(search_e_prime_i),search_e__is__elem_tllign).
    ↪ formula(elem(search_e_i),search_e__is__elem_tllign).
formula(elem(search_e_prime_i),search_e__is__elem_tllign).
    ↪ formula(elem(search_e_i),search_e__is__elem_tllign).
formula(elem(search_e_prime_i),search_e__is__elem_tllign).
    ↪ formula(elem(search_e_i),search_e__is__elem_tllign).
formula(addr(remove_prev_prime_i),remove_prev__is__addr_tllign).
    ↪     formula(addr(remove_prev_i),remove_prev__is__addr_tllign
    ↪ ).
formula(addr(remove_curr_prime_i),remove_curr__is__addr_tllign).
    ↪     formula(addr(remove_curr_i),remove_curr__is__addr_tllign
    ↪ ).
formula(addr(remove_aux_prime_i),remove_aux__is__addr_tllign).
    ↪ formula(addr(remove_aux_i),remove_aux__is__addr_tllign).
formula(elem(remove_e_prime_i),remove_e__is__elem_tllign).
    ↪ formula(elem(remove_e_i),remove_e__is__elem_tllign).
formula(elem(remove_e_prime_i),remove_e__is__elem_tllign).
    ↪ formula(elem(remove_e_i),remove_e__is__elem_tllign).
formula(elem(remove_e_prime_i),remove_e__is__elem_tllign).
    ↪ formula(elem(remove_e_i),remove_e__is__elem_tllign).
formula(addr(insert_prev_prime_i),insert_prev__is__addr_tllign).
    ↪     formula(addr(insert_prev_i),insert_prev__is__addr_tllign
    ↪ ).
formula(addr(insert_curr_prime_i),insert_curr__is__addr_tllign).
    ↪     formula(addr(insert_curr_i),insert_curr__is__addr_tllign
    ↪ ).
formula(addr(insert_aux_prime_i),insert_aux__is__addr_tllign).
    ↪ formula(addr(insert_aux_i),insert_aux__is__addr_tllign).
formula(elem(insert_e_prime_i),insert_e__is__elem_tllign).
    ↪ formula(elem(insert_e_i),insert_e__is__elem_tllign).
formula(elem(insert_e_prime_i),insert_e__is__elem_tllign).
    ↪ formula(elem(insert_e_i),insert_e__is__elem_tllign).
formula(elem(insert_e_prime_i),insert_e__is__elem_tllign).
    ↪ formula(elem(insert_e_i),insert_e__is__elem_tllign).


%%%% Type equivalences:::::
formula(forall([mem(x)],and(     not(path(x)),not(addr(x)),not(
    ↪ setaddr(x)),not(elem(x)),not(setelem(x)),not(cell(x)),not(
```

```
    ↪ nat(x)),not(tid(x)),not(settid(x)))),
    ↪ mem_is_not_other_type_tllign).
formula(forall([path(x)],and(    not(mem(x)),not(addr(x)),not(
    ↪ setaddr(x)),not(elem(x)),not(setelem(x)),not(cell(x)),not(
    ↪ nat(x)),not(tid(x)),not(settid(x)))),
    ↪ path_is_not_other_type_tllign).
formula(forall([addr(x)],and(    not(mem(x)),not(path(x)),not(
    ↪ setaddr(x)),not(elem(x)),not(setelem(x)),not(cell(x)),not(
    ↪ nat(x)),not(tid(x)),not(settid(x)))),
    ↪ addr_is_not_other_type_tllign).
formula(forall([setaddr(x)],and(not(mem(x)),not(path(x)),not(addr
    ↪ (x)),not(elem(x)),not(setelem(x)),not(cell(x)),not(nat(x)),
    ↪ not(tid(x)),not(settid(x)))),
    ↪ setaddr_is_not_other_type_tllign).
formula(forall([elem(x)],and(    not(mem(x)),not(path(x)),not(addr
    ↪ (x)),not(setaddr(x)),not(setelem(x)),not(cell(x)),nat(x),
    ↪ not(tid(x)),not(settid(x)))),elem_is_not_other_type_tllign)
    ↪ .
formula(forall([setelem(x)],and(not(mem(x)),not(path(x)),not(addr
    ↪ (x)),not(setaddr(x)),not(elem(x)),not(cell(x)),not(nat(x)),
    ↪ not(tid(x)),not(settid(x)))),
    ↪ setelem_is_not_other_type_tllign).
formula(forall([cell(x)],and(    not(mem(x)),not(path(x)),not(addr
    ↪ (x)),not(setaddr(x)),not(elem(x)),not(setelem(x)),not(nat(x
    ↪ )),not(tid(x)),not(settid(x)))),
    ↪ cell_is_not_other_type_tllign).
formula(forall([nat(x)],and(    not(mem(x)),not(path(x)),not(addr
    ↪ (x)),not(setaddr(x)),not(setelem(x)),not(cell(x)),not(
    ↪ settid(x)))),nat_is_not_other_type_tllign).
formula(forall([tid(x)],and(    not(mem(x)),not(path(x)),not(addr
    ↪ (x)),not(setaddr(x)),not(elem(x)),not(setelem(x)),not(cell(
    ↪ x)),nat(x),not(settid(x)))),tid_is_not_other_type_tllign).
formula(forall([settid(x)],and(not(mem(x)),not(path(x)),not(addr(
    ↪ x)),not(setaddr(x)),not(elem(x)),not(setelem(x)),not(cell(x
    ↪ )),not(nat(x)),not(tid(x)))),
    ↪ settid_is_not_other_type_tllign).
```

%%%% Set

```
formula(forall([setaddr(se),setaddr(se2),addr(x)],equiv(or(in(x,
    ↪ se),in(x,se2)),in(x,union(se,se2)))),union_def).
formula(forall([setaddr(se),setaddr(se2)],equal(union(se,se2),
    ↪ union(se2,se))),union_conmutative).
```

```
formula(forall([addr(b),addr(a),setaddr(se)],implies(not(in(a,se)
    ↪ ),implies(in(b,se),not(equal(b,a))))),in_set__def).
formula(forall([addr(a),addr(b)],and(implies(not(equal(a,b)),not(
    ↪ in(b,singl(a))))  ,in(a,singl(a)))),a__in__singl_a).
formula(forall([addr(a)],not(in(a,emptySet))),emptySet_is_empty).
formula(forall([setaddr(se1),setaddr(se2)],equiv(not(exists([addr
    ↪ (a)],equiv(in(a,se1),not(in(a,se2))))),equal(se1,se2))),
    ↪ set_eq_addr).
formula(forall([setaddr(se1),setaddr(se2)],implies(equal(se1,se2)
    ↪ ,forall([addr(a)],equiv(in(a,se1),in(a,se2))))),
    ↪ set_extenaddr).
formula(forall([setaddr(se1),setaddr(se2)],implies(forall([addr(a
    ↪ )],equiv(in(a,se1),in(a,se2))),     equal(se1,se2))),
    ↪ set_exten_invaddr).
formula(forall([setaddr(se),setaddr(se2),addr(x)],equiv(and(in(x,
    ↪ se),not(in(x,se2))),in(x,setDiff(se,se2)))),SetDiff_def).
formula(forall([addr(a),setaddr(se)],implies(in(a,se),not(in(a,
    ↪ setDiff(se,singl(a)))))),a_not__in_se_dif_a).
```

```
%%%% Cell
formula(forall([cell(c)],exists([elem(e),addr(a),tid(t)],equal(c,
    ↪ mkcell(e,a,t)))),mckcell__def_tllign).
formula(forall([elem(e),addr(a),tid(t)],equal(data(mkcell(e,a,t))
    ↪ ,e)),data__def).
formula(forall([elem(e),addr(a),tid(t)],equal(next(mkcell(e,a,t))
    ↪ ,a)),next__def).
formula(forall([elem(e),addr(a),tid(t)],equal(lockid(mkcell(e,a,t
    ↪ )),t)),lockid__def).
formula(equal(next(error),null),next_error__is__null).
formula(forall([cell(c1),cell(c2)],implies(equal(c1,c2),and(equal
    ↪ (data(c1),data(c2)),equal(lockid(c1),lockid(c2)),equal(next
    ↪ (c1),next(c2))))),equality_bt_cell).
formula(forall([mem(m),addr(a),addr(b)],implies(equal(a,b),equal(
    ↪ rd(m,a),rd(m,b)))),equality_on_read).
```

```
        %%%% mem
formula(forall([mem(m),addr(a),addr(b),cell(c),mem(m2)],implies(
    ↪ not(equal(a,null)),implies(equal(upd(m,a,c),m2),equal(rd(m2
    ↪ ,a),c)))),upd__def__not_null).
formula(forall([mem(m),addr(a),addr(b),cell(c),mem(m2)],implies(
    ↪ and(not(equal(a,null)),not(equal(a,b))),implies(equal(upd(m
    ↪ ,a,c),m2),equal(rd(m,b),rd(m2,b))))),
    ↪ upd__def__one_at_the_time).
formula(forall([mem(m)],equal(rd(m,null),error)),rd_mem__def).
```

```
        %%%%% elem
formula(not(equal(lowestElem,highestElem)),
    ↪ lowest__less_than_highest).
formula(forall([elem(e)],or(equal(e,lowestElem),ls_elem(
    ↪ lowestElem,e))),lowestElem__def_tll).
formula(forall([elem(e)],or(equal(e,highestElem),ls_elem(e,
    ↪ highestElem))),highestElem__def_tll).
formula(forall([elem(x),elem(y),elem(z)],implies(and(ls_elem(x,y)
    ↪ ,ls_elem(y,z)),ls_elem(x,z))),less_trans).
formula(forall([elem(x),elem(y)],not(and(ls_elem(x,y),ls_elem(y,x
    ↪ )))),less_total).
formula(forall([elem(x),elem(y)],equiv(ls_elem(x,y),and(not(equal
    ↪ (x,y)),not(ls_elem(y,x))))),ls_xy__not_ls_yx).
%%%%%% Important axioms:

formula(forall([addr(c),addr(a),addr(b),mem(m),setaddr(se)],
    ↪ implies(and(in(a,se),equal(se,addr2set(m,b)),equal(c,next(
    ↪ rd(m,a))),not(equal(a,null))),in(c,se))),nextreg).
formula(forall([tid(t),mem(hp_p),mem(hp),addr(a),addr(hd)],
    ↪ implies(and(equal(hp_p,upd(hp,a,mkcell(data(rd(hp,a)),next(
    ↪ rd(hp,a)),t)))),equal(addr2set(hp,hd),addr2set(hp_p,hd)))),
    ↪ lock_keeps_addr2set).
formula(forall([tid(t),mem(hp_p),mem(hp),addr(tl),addr(a),addr(hd
    ↪ )],implies(and(equal(hp_p,upd(hp,a,mkcell(data(rd(hp,a)),
    ↪ next(rd(hp,a)),t)))),equiv(orderlist(hp,hd,tl),orderlist(
    ↪ hp_p,hd,tl))))),lock_keeps_orderlist).
formula(forall([tid(t),mem(hp_p),mem(hp),addr(a),addr(hd)],
    ↪ implies(and(equal(hp_p,upd(hp,a,mkcell(data(rd(hp,a)),next(
    ↪ rd(hp,a)),t)))),equal(data(rd(hp,a)),data(rd(hp_p,a))))),
    ↪ lock_keeps_heap__data).
formula(forall([tid(t),mem(hp_p),mem(hp),addr(a),addr(hd)],
    ↪ implies(and(equal(hp_p,upd(hp,a,mkcell(data(rd(hp,a)),next(
    ↪ rd(hp,a)),t)))),equal(next(rd(hp,a)),next(rd(hp_p,a))))),
    ↪ lock_keeps_heap__next).
formula(forall([addr(a),addr(hd),cell(c),mem(hp)],implies(and(not
    ↪ (in(a,addr2set(hp,hd)))),equal(addr2set(hp,hd),addr2set(upd
    ↪ (hp,a,c),hd)))),not_in_region__not_change_heap_addr).
formula(forall([addr(tl),addr(a),addr(hd),cell(c),mem(hp)],
    ↪ implies(and(not(in(a,addr2set(hp,hd)))),equiv(orderlist(hp,
    ↪ hd,tl),orderlist(upd(hp,a,c),hd,tl)))),
    ↪ not_in_region__not_change_heap_list).
formula(forall([mem(hp),addr(hd),addr(tl),addr(nl)],implies(and(
    ↪ ls_elem(data(rd(hp,hd)),data(rd(hp,tl))),equal(next(rd(hp,
    ↪ hd)),tl),equal(next(rd(hp,tl)),nl)),orderlist(hp,hd,tl))),
    ↪ order_primim).
formula(forall([mem(hp),addr(hd),addr(tl)],implies(and(ls_elem(
    ↪ data(rd(hp,hd)),data(rd(hp,tl))),equal(next(rd(hp,hd)),tl),
    ↪ equal(next(rd(hp,tl)),null)),equal(addr2set(hp,hd),union(
```

```
    ↪ union(singl(hd),singl(tl)),singl(null))))),addr2set_primim)
    ↪ .
formula(forall([addr(hd),addr(prev),addr(aux),addr(curr),mem(hp),
    ↪ mem(hp_p),setaddr(reg),setaddr(reg_p)],implies(and(equal(
    ↪ reg,addr2set(hp,hd)),equal(union(reg,singl(aux)),reg_p),
    ↪ equal(next(rd(hp,prev)),curr),not(equal(prev,curr)),equal(
    ↪ next(rd(hp,aux)),curr),not(equal(aux,null)),not(equal(prev,
    ↪ null)),not(equal(curr,null)),in(prev,addr2set(hp,hd)),equal
    ↪ (hp_p,upd(hp,prev,mkcell(data(rd(hp,prev)),aux,lockid(rd(hp
    ↪ ,prev))))))),equal(reg_p,addr2set(hp_p,hd)))),
    ↪ insert__keeps_addr2set).
formula(forall([addr(curr),addr(aux),addr(prev),addr(hd),mem(hp),
    ↪ mem(hp_p)],implies(and(equal(next(rd(hp,curr)),aux),equal(
    ↪ next(rd(hp,prev)),curr),not(equal(aux,next(rd(hp,prev))))),
    ↪ equal(hp_p,upd(hp,prev,mkcell(data(rd(hp,prev)),aux,lockid(
    ↪ rd(hp,prev))))),not(equal(aux,null)),in(curr,addr2set(hp,hd
    ↪ )),in(null,addr2set(hp,hd)),in(prev,addr2set(hp,hd))),equal
    ↪ (setDiff(addr2set(hp,hd),singl(curr)),addr2set(hp_p,hd)))),
    ↪ remove__keeps_addr2set).
%%% Addr2set
formula(forall([mem(m),addr(a)],equal(addr2set(m,a),union(singl(a
    ↪ ),addr2set(m,next(rd(m,a)))))),addr2set_rec_def).
formula(forall([mem(m)],equal(addr2set(m,null),singl(null))),
    ↪ addr2set_null__is__singl_null).
```

```
%%% Orderlist
formula(forall([addr(a),addr(b),addr(c),addr(d),addr(hd),addr(tl)
    ↪ ,mem(hp)], implies( and( orderlist(hp,hd,tl),in(a,addr2set(
    ↪ hp,hd)) , in(b,addr2set(hp,hd)) , in(c,addr2set(hp,hd)) ,
    ↪ in(d,addr2set(hp,hd)),not(equal(tl,null)),equal(null,next(
    ↪ rd(hp,tl))),not(equal(c,null)),not(equal(d,null)),not(equal
    ↪ (a,null)),not(equal(b,null)),not(equal(a,tl)),not(equal(b,
    ↪ tl)),equal(next(rd(hp,c)),a),equal(next(rd(hp,d)),b)),
    ↪ implies(equal(a,b),equal(c,d)))),next_injective__if_ordered
    ↪ ).
formula(forall([addr(a),addr(tl),addr(hd),mem(hp)],implies(and(in
    ↪ (a,addr2set(hp,hd)),in(tl,addr2set(hp,hd)),not(equal(hd,
    ↪ null)),not(equal(tl,null)),not(equal(a,null)),equal(next(rd
    ↪ (hp,tl)),null)),not(equal(next(rd(hp,a)),a)))),
    ↪ next_is_not_same__if_ordered).
formula(forall([addr(d),addr(tl),addr(hd),mem(hp)],implies(and(in
    ↪ (tl,addr2set(hp,hd)),not(equal(hd,null)),not(equal(tl,null)
    ↪ ),equal(next(rd(hp,tl)),null),not(equal(d,null)),equal(next
    ↪ (rd(hp,d)),null),in(d,addr2set(hp,hd))),equal(d,tl))),
    ↪ just_tail__points__null).
```

```
    formula(forall([addr(hd),mem(hp),mem(hp_p),addr(aux),addr(prev),
        ↪ addr(curr),addr(tl)],implies(and(orderlist(hp,hd,tl),not(
        ↪ equal(tl,null)),equal(next(rd(hp,tl)),null),ls_elem(data(rd
        ↪ (hp,prev)),data(rd(hp,aux))),ls_elem(data(rd(hp,aux)),data(
        ↪ rd(hp,curr))),equal(next(rd(hp,aux)),curr),equal(next(rd(hp
        ↪ ,prev)),curr),equal(hp_p,upd(hp,prev,mkcell(data(rd(hp,prev
        ↪ )),aux,lockid(rd(hp,prev))))),orderlist(hp_p,hd,tl))),
        ↪ insert__keeps_orderlist).
    formula(forall([addr(hd),mem(hp),mem(hp_p),addr(aux),addr(prev),
        ↪ addr(curr),addr(tl)],implies(and(equal(aux,next(rd(hp,curr)
        ↪ )),equal(curr,next(rd(hp,prev))),not(equal(aux,null)),equal
        ↪ (null,next(rd(hp,tl))),not(equal(aux,next(rd(hp,prev)))),
        ↪ equal(hp_p,upd(hp,prev,mkcell(data(rd(hp,prev)),aux,lockid(
        ↪ rd(hp,prev))))),in(prev,addr2set(hp,hd)),in(curr,addr2set(
        ↪ hp,hd)),in(null,addr2set(hp,hd)),in(aux,addr2set(hp,hd)),
        ↪ orderlist(hp,hd,tl)),orderlist(hp_p,hd,tl))),
        ↪ remove__keeps_orderlist).

end_of_list.


list_of_formulae(conjectures).
    formula(false).
end_of_list.


end_problem.
```

C

Code

The code is shown with annotations and not with line numbers in order to better understand the invariants. These annotations are used to express the invariants. The lines of code wrapped by ':tag[' and ':tag] ' are used to define the preconditions of the invariants. In stead of: *(pc(i)* $\leq n \wedge$ *pc(i)* $\geq m \rightarrow$ *...)* ; it is written: *(@tag* $\rightarrow$ *)*. This way is clearer to read the invariants within the code.

We consider *head* and *tail* sentinel nodes which are neither removed nor modified and we assume that the list is initialized with *head* and *tail* already set. The set *reg* is initialized containing solely the addresses of *head* and *tail*. Similarly, the set *elems* is initialized containing only the elements initially stored at the nodes pointed by *head* and *tail*. There is also a function **havocListElem**() which returns a random element.

```
global
   addr   head
   addr   tail
   ghost   addrSet   region
   ghost   elemSet   elements

assume
   region   =   union(union({head},{tail}),{null})   ∧
   data(rd(heap,   head))   =   lowestElem   ∧
   data(rd(heap,   tail))   =   highestElem   ∧
   head   !=   tail   ∧
   head   !=   null   ∧
   tail   !=   null   ∧
   head→next  =   tail   ∧
   tail→next  =   null


//   -----   PROGRAM   BEGINS   ------------------------------------

                procedure   main   ()
                   elem   e
                begin
```

```
':main\_body['       while   (true)   do
                        //   Generate   random   e
                        e   :=   havocListElem();
':main\_e['             choice
                           call   search(e);
                           _or_
                           call   insert(e);
                           _or_
                           call   remove(e);
                        endchoice
                     endwhile
                     return   ();
':main\_e]'
':main\_body]'    end

//   -----   SEARCH   ------------------------------------------------


                  procedure   search   (e:elem)   :   bool
                     addr   prev
                     addr   curr
                     addr   aux
                     bool   result

                  begin
':search\_body['          prev   :=   head;
':sch\_prev\_lower['
':sch\_prev\_def['
':sch\_prev\_is\_head['   prev→lock;
':sch\_owns\_prev['       curr   :=   prev→next;
':sch\_curr\_def['
':sch\_follows['          curr→lock;
':sch\_prev\_is\_head]'
':sch\_owns\_curr\_one['  while   (curr→data  <   e)   do
':sch\_while\_begins['        aux   :=   prev;
':sch\_aux\_eq\_prev'         prev   :=   curr;
':sch\_equals['
':sch\_aux\_before\_prev'     aux→unlock;
                              curr   :=   curr→next;
':sch\_while\_begins]'
':sch\_equals]'
':sch\_owns\_curr\_one]'      curr→lock;
':sch\_owns\_curr\_two['   endwhile
':sch\_after\_lookup'      result   :=   curr→data  =   e;
':sch\_prev\_lower]'
':sch\_result\_set['
':sch\_diff['              prev→unlock;
':sch\_owns\_prev]'
':sch\_follows]'
':sch\_prev\_def]'         curr→unlock;
':sch\_owns\_curr\_two]'
':sch\_diff]'
':sch\_curr\_def]'         return   (result);
':sch\_result\_set]'
':search\_body]'      end

//   -----   INSERT   ------------------------------------------------


                  procedure   insert   (e:elem)
                     addr   prev
```

```
                    addr   curr
                    addr   aux

                begin
':insert\_body['
':ins\_head\_next\_diff['   prev   :=   head;
':ins\_prev\_lower['
':ins\_prev\_def['
':ins\_prev\_is\_head['     prev→lock;
':ins\_owns\_prev['         curr   :=   prev→next;
':ins\_head\_next\_diff]'
':ins\_curr\_def['
':ins\_follows['            curr→lock;
':ins\_prev\_is\_head]'
':ins\_owns\_curr\_one['
':ins\_lookup\_loop['
':ins\_lookup\_condition'   while   (curr→data  <   e)   do
':ins\_while\_begins['
':ins\_while['                aux   :=   prev;
':ins\_aux\_eq\_prev'         prev   :=   curr;
':ins\_equals['
':ins\_aux\_before\_prev'     aux→unlock;
                              curr   :=   curr→next;
':ins\_while\_begins]'
':ins\_equals]'
':ins\_while]'
':ins\_owns\_curr\_one]'      curr→lock;
':ins\_owns\_curr\_two['   endwhile
':ins\_lookup\_loop]'
':ins\_final\_conditional' if   (curr   !=   null   ∧  curr→data  >   e)   then
':ins\_insert['                aux   :=   malloc(e,   null,   #);
':after\_malloc['              aux→next  :=   curr;
':ins\_aux\_before\_curr'
':ins\_diff['                  prev→next  :=   aux
                               region   :=   union   (region,   {aux});
':ins\_follows]'
':after\_malloc]'
':ins\_insert]'
':ins\_prev\_lower]'       endif
':ins\_elem\_inserted['    prev→unlock;
':ins\_owns\_prev]'
':ins\_prev\_def]'         curr→unlock;
':ins\_owns\_curr\_two]'
':ins\_curr\_def]'
':ins\_diff]'           return();
':ins\_elem\_inserted]'
':insert\_body]'      end

//   -----   REMOVE   ------------------------------------------------


                procedure   remove   (e:elem)
                    addr   prev
                    addr   curr
                    addr   aux

                begin
':remove\_body['          prev   :=   head;
':rem\_prev\_lower['
':rem\_prev\_def['
':rem\_prev\_is\_head['   prev→lock;
```

```
':rem\_owns\_prev['       curr    :=    prev→next;
':rem\_curr\_def['
':rem\_follows['          curr→lock;
':rem\_prev\_is\_head]'
':rem\_owns\_curr\_one['
':rem\_lookup\_loop['     while    (curr→data  <    e)    do
':rem\_while\_begins['
':rem\_while['                 aux    :=    prev;
':rem\_aux\_eq\_prev'          prev    :=    curr;
':rem\_equals['
':rem\_aux\_before\_prev'      aux→unlock;
                               curr    :=    curr→next;
':rem\_while\_begins]'
':rem\_equals]'
':rem\_while]'
':rem\_owns\_curr\_one]'       curr→lock;
':rem\_owns\_curr\_two['    endwhile
':rem\_lookup\_loop]'
':rem\_final\_conditional'  if    (curr→data  =    e)    then
':rem\_remove['
':rem\_if\_one'                  aux    :=    curr→next;
':rem\_if\_two'                  prev→next  :=    aux
                        /*
                                  region    :=    diff(region,    {curr});
                        */
':rem\_follows]'
':rem\_curr\_def]'
':rem\_remove]'
':rem\_prev\_lower]'        endif
':rem\_elem\_removed['
':rem\_diff['               prev→unlock;
':rem\_owns\_prev]'
':rem\_prev\_def]'          curr→unlock;
':rem\_diff]'
':rem\_owns\_curr\_two]'    return();
':rem\_elem\_removed]'
':remove\_body]'        end
```

D

Invariants

Preserve

```
vars:


invariant [preserve] :

    in (null, region)

#region:
    region = addr2set(heap, head)


    rd(heap, tail).next = null
    tail != null

#head_not_null:
    head != null

#head_not_tail:
    head != tail

#elements:
    rd(heap, head).data = lowestElem
    rd(heap, tail).data = highestElem

#ordered:
    orderlist (heap, head, tail)

    in (tail, region)
```

Disjoint

```
vars:

tid i
tid j

invariant [disjoint] :

(i != j ∧ \@after_malloc(i). ∧ \@after_malloc(j).) → insert::aux(i) !=
    ↪ insert::aux(j)
```

Order

```
vars:

tid i

invariant [order] :

   rd(heap, head).data = lowestElem
   rd(heap, tail).data = highestElem

   @main_e(i). → (main::e(i) != lowestElem ∧ main::e(i) != highestElem)
   @search_body(i). → (search::e(i) != lowestElem ∧ search::e(i) !=
      ↪ highestElem)

#insert_bounded_e:
   @insert_body(i). → (insert::e(i) != lowestElem ∧ insert::e(i) !=
      ↪ highestElem)

#remove_bounded_e:
   @remove_body(i). → (remove::e(i) != lowestElem ∧ remove::e(i) !=
      ↪ highestElem)



///////  Search   ////////////////////////


#search_curr_bounded:
   @sch_after_lookup(i). →
      (rd(heap,search::curr(i)).data > search::e(i) ∨
       rd(heap,search::curr(i)).data = search::e(i))

   @sch_prev_def(i). → (rd(heap, search::prev(i)).data < rd(heap, tail).
      ↪ data ∨
                          rd(heap, search::prev(i)).data = rd(heap, tail).
                             ↪ data)
   @sch_while_begins(i). → rd(heap, search::curr(i)).data < search::e(i)

#search_prev_lower:
   @sch_prev_lower(i). → rd(heap, search::prev(i)).data < search::e(i)
```

```
///////  Insert  ////////////////////////

   @ins_curr_def(i). → (rd(heap, insert::curr(i)).data < rd(heap, tail).
      ↪ data ∨
                          rd(heap, insert::curr(i)).data = rd(heap, tail).
                             ↪ data)
   @ins_prev_def(i). → (rd(heap, insert::prev(i)).data < rd(heap, tail).
      ↪ data ∨
                          rd(heap, insert::prev(i)).data = rd(heap, tail).
                             ↪ data)
   @ins_while_begins(i). → rd(heap, insert::curr(i)).data < insert::e(i)

#insert_prev_lower:
   @ins_prev_lower(i). → rd(heap, insert::prev(i)).data < insert::e(i)

#insert_curr_higher:
   @ins_insert(i). → rd(heap, insert::curr(i)).data > insert::e(i)

#insert_aux_is_e:
   @after_malloc(i). → rd(heap, insert::aux(i)).data = insert::e(i)

#insert_curr_bounded:
   @ins_final_conditional(i). →
      (rd(heap,insert::curr(i)).data > insert::e(i) ∨
       rd(heap,insert::curr(i)).data = insert::e(i))


///////  Remove  ////////////////////////


#remove_curr_is_e:
   @rem_remove(i). → rd(heap, remove::curr(i)).data = remove::e(i)

#remove_curr_bounded:
   @rem_final_conditional(i). →
      (rd(heap,remove::curr(i)).data > remove::e(i) ∨
       rd(heap,remove::curr(i)).data = remove::e(i))

   @rem_prev_def(i). → (rd(heap, remove::prev(i)).data < rd(heap, tail).
      ↪ data ∨
                          rd(heap, remove::prev(i)).data = rd(heap, tail).
                             ↪ data)
   @rem_while_begins(i). → rd(heap, remove::curr(i)).data < remove::e(i)

#remove_prev_lower:
   @rem_prev_lower(i). → rd(heap, remove::prev(i)).data < remove::e(i)
```

Locks

```
vars:

tid i
```

```
invariant [locks] :

#search_owns_prev:
   (@sch_owns_prev(i). → rd(heap, search::prev(i)).lockid = i)

#search_owns_curr:
   ((@sch_owns_curr_one(i). ∨ @sch_owns_curr_two(i).) → rd(heap, search::
      ↪ curr(i)).lockid = i)

   (@sch_aux_before_prev(i). → rd(heap, search::aux(i)).lockid = i)

#insert_owns_prev:
   (@ins_owns_prev(i). → rd(heap, insert::prev(i)).lockid = i)

#insert_owns_curr:
   ((@ins_owns_curr_one(i). ∨ @ins_owns_curr_two(i).) → rd(heap, insert::
      ↪ curr(i)).lockid = i)

   (@ins_aux_before_prev(i). → rd(heap, insert::aux(i)).lockid = i)


   (@rem_owns_prev(i). → rd(heap, remove::prev(i)).lockid = i)

#remove_owns_curr:
   ((@rem_owns_curr_one(i). ∨ @rem_owns_curr_two(i).) → rd(heap, remove::
      ↪ curr(i)).lockid = i)

   (@rem_aux_before_prev(i). → rd(heap, remove::aux(i)).lockid = i)
```

Region

```
vars:

tid i

invariant [region] :


   in (head, region)
   in (tail, region)
   in (null, region)



///////  Search   ////////////////////////


#search_prev_in_region:
   @sch_prev_def(i). → in (search::prev(i), region)

#search_curr_in_region:
   @sch_curr_def(i). → in (search::curr(i), region)
```

```
    @sch_aux_before_prev(i). → in (search::aux(i), region)


///////  Insert   ////////////////////////


#insert_prev_in_region:
    @ins_prev_def(i). → in (insert::prev(i), region)

#insert_curr_in_region:
    @ins_curr_def(i). → in (insert::curr(i), region)

#insert_aux_not_in_region:
    @after_malloc(i). → ~ (in (insert::aux(i), region))

    @ins_aux_before_prev(i). → (in (insert::aux(i), region))


///////  Remove   ////////////////////////


#remove_prev_not_null:
    @rem_prev_def(i). → remove::prev(i) != null

#remove_prev_in_region:
    @rem_prev_def(i). → in (remove::prev(i), region)

#remove_curr_in_region:
    @rem_curr_def(i). → in (remove::curr(i), region)

    @rem_aux_before_prev(i). → in (remove::aux(i), region)
```

Next

```
vars:

tid j

invariant [nexts] :


    rd(heap, head).next != head
    head != tail // Same as in remove_region
    tail != null // Same as in remove_region
    rd(heap, head).next != null



///////  Search   ////////////////////////


    @sch_prev_def(j). → search::prev(j) != null
    @sch_prev_is_head(j). → (search::prev(j) = head ∧ rd(heap, head).next !=
        ↪  null)
```

```
    @sch_aux_eq_prev(j). → search::aux(j) = search::prev(j)
    @sch_equals(j). → search::prev(j) = search::curr(j)
    @sch_aux_before_prev(j). → (search::aux(j) != search::prev(j) ∧ rd(heap,
        ↪  search::aux(j)).next = search::prev(j))

#search_prev_next_curr:
    ((@sch_follows(j). ∧ ~ @sch_equals(j).) → (search::prev(j) != search::
        ↪ curr(j) ∧
                                        rd(heap,search::prev(j)).next = search
                                            ↪ ::curr(j)))

    @sch_follows(j). → search::curr(j) != null
    @sch_diff(j). → search::prev(j) != search::curr(j)



///////  Insert   ////////////////////////


     @ins_head_next_diff(j). → rd(heap, head).next != head
     @ins_prev_is_head(j). → insert::prev(j) = head
     @ins_diff(j). → insert::prev(j) != insert::curr(j)
     (@ins_aux_eq_prev(j). → insert::aux(j) = insert::prev(j))

#insert_aux_next_prev:
    (@ins_aux_before_prev(j). → (insert::aux(j) != insert::prev(j) ∧ rd(heap
        ↪ , insert::aux(j)).next = insert::prev(j)))

#insert_aux_next_curr:
    @ins_aux_before_curr(j). → (rd(heap,insert::aux(j)).next = insert::curr(
        ↪ j))

#insert_prev_next_curr:
     (@ins_follows(j). ∧ ~ @ins_equals(j).) → (insert::prev(j) != insert::
         ↪ curr(j) ∧
                                             rd(heap,insert::prev(j)).next =
                                                 ↪ insert::curr(j))

     @ins_equals(j). → insert::prev(j) = insert::curr(j)
    (@ins_prev_is_head(j).  @ins_lookup_loop(j).) → insert::prev(j) != null

#insert_curr_not_null:
    @ins_follows(j). → insert::curr(j) != null

    @ins_while_begins(j). → rd(heap, insert::curr(j)).next != null



///////  Remove   ////////////////////////


    @rem_prev_is_head(j). → (remove::prev(j) = head ∧ rd(heap, head).next !=
        ↪  null)
    (@rem_aux_eq_prev(j). → remove::aux(j) = remove::prev(j))
    (@rem_equals(j). → remove::prev(j) = remove::curr(j))
    (@rem_aux_before_prev(j). → (remove::aux(j) != remove::prev(j) ∧ rd(heap
```

```
      ↪ , remove::aux(j)).next = remove::prev(j)))

#remove_prev_next_curr:
   ((@rem_follows(j). ∧ ~ @rem_equals(j).) → (remove::prev(j) != remove::
      ↪ curr(j) ∧
                                          rd(heap,remove::prev(j)).next = remove
                                             ↪ ::curr(j)))

   @rem_diff(j). → remove::prev(j) != remove::curr(j)

#remove_curr_next_aux:
   @rem_if_two(j). → ( rd(heap, remove::curr(j)).next = remove::aux(j)
                       ∧ remove::prev(j) != remove::aux(j)
                       ∧ remove::curr(j) != remove::aux(j))

   (@rem_prev_is_head(j). @rem_lookup_loop(j).) → remove::prev(j) != null

#remove_curr_not_null:
   @rem_follows(j). → remove::curr(j) != null

   @rem_while_begins(j). → rd(heap, remove::curr(j)).next != null
```